

1. Preface

1. [Dsp00095-Preface to Digital Signal Processing - DSP](#)

2. Time Series

1. [Dsp00100-Periodic Motion and Sinusoids](#)
2. [Dsp00104-Sampled Time Series](#)
3. [Dsp00108-Averaging Time Series](#)

3. Plotting Data

1. [Java1468-Plotting Engineering and Scientific Data using Java](#)
2. [Java1489-Plotting 3D Surfaces using Java](#)
3. [Java1492-Plotting Large Quantities of Data using Java](#)

4. Fourier Analysis

1. [Java1478-Fun with Java, How and Why Spectral Analysis Works](#)
2. [Java1482-Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#)
3. [Java1483-Spectrum Analysis using Java, Frequency Resolution versus Data Length](#)
4. [Java1484-Spectrum Analysis using Java, Complex Spectrum and Phase Angle](#)
5. [Java1485-Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain](#)
6. [Java1486-Fun with Java, Understanding the Fast Fourier Transform \(FFT\) Algorithm](#)
7. [Java1490-2D Fourier Transforms using Java, Part 1](#)
8. [Java1491-2D Fourier Transforms using Java, Part 2](#)

5. Convolution and Frequency Filtering

1. [Java1487-Convolution and Frequency Filtering in Java](#)
2. [Java1488-Convolution and Matched Filtering in Java](#)

6. Adaptive Processing

1. [Java2350-Adaptive Filtering in Java, Getting Started](#)

2. [Java2352-An Adaptive Whitening Filter in Java](#)
3. [Java2354-A General-Purpose LMS Adaptive Engine in Java](#)
4. [Java2356-An Adaptive Line Tracker in Java](#)
5. [Java2358-Adaptive Identification and Inverse Filtering using Java](#)
6. [Java2360-Adaptive Noise Cancellation using Java](#)
7. [Java2362-Adaptive Prediction using Java](#)
7. Data Compression and the Discrete Cosine Transform
 1. [Java2440 Understanding the Lempel-Ziv Data Compression Algorithm in Java](#)
 2. [Java2442 Understanding the Huffman Data Compression Algorithm in Java](#)
 3. [Java2444 Understanding the Discrete Cosine Transform in Java](#)
 4. [Java2446 Understanding the 2D Discrete Cosine Transform in Java](#)
 5. [Java2448 Understanding the 2D Discrete Cosine Transform in Java, Part 2](#)

Dsp00095-Preface to Digital Signal Processing - DSP

This module is a preface to the collection titled Digital Signal Processing-DSP

Revised: Fri Oct 16 23:06:26 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

Over the years, I have published a large number of DSP tutorials on various websites. This collection, which is a work in process, gathers the more significant of those tutorials into a common location to make them more readily available for Connexions users.

Some of the tutorials were originally published ten or more years ago. However, you need not be concerned about the material in those tutorials becoming obsolete. The concepts and algorithms involved in DSP (*such as convolution, correlation, the discrete Fourier transform and the Fast Fourier Transform*) are essentially the same today as they were when they first became practical for use on digital computers around 1960. (*However, the hardware used to implement those algorithms has become much smaller and much faster.*)

As I have time available, I am converting the tutorials from their original HTML format into the Openstax format that you are accustomed to seeing. You will find that some of the tutorials are available in Openstax format, and others are available in HTML or PDF format as described below.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of October 8, 2015, some of the functionality of the legacy presentation format that is required by modules in this collection have not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in HTML versions of the tutorials may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF versions of the tutorials using the PDF links that are provided.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorials in their original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF versions or the HTML versions directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Dsp00095-Preface to Digital Signal Processing-DSP

- File: Dsp00095.htm
- Published: 04/11/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Dsp00100-Periodic Motion and Sinusoids

Baldwin kicks off a new miniseries on DSP. He discusses periodic motion and sinusoids. He introduces time series analysis, sine and cosine functions, and frequency decomposition. He discusses composition, and provides examples for square and triangular waveforms.

Revised: Fri Oct 16 23:09:45 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Periodic motion](#)
 - [Motion of a pendulum](#)
 - [A bag of sand](#)
 - [Harmonic motion](#)
 - [Sinusoids](#)
 - [Separate the cosine and sine components](#)
 - [Plotting separate sine and cosine functions](#)

- [Time varying functions](#)
- [Sampled data](#)
- [The period of the sine and cosine functions](#)
 - [Different periods or frequencies](#)
 - [Modified arguments](#)
 - [Why does pi appear in the arguments?](#)
 - [Radians versus cycles](#)
 - [The horizontal scale](#)
 - [The argument to the sine and cosine functions](#)
 - [Where does \$\sin\(\arg\)\$ equal zero?](#)
 - [Where are the peaks in the cosine function?](#)
- [Composition and decomposition](#)
 - [An approximate square waveform](#)
 - [Successive approximations](#)
 - [An approximate triangular waveform](#)
- [So what?](#)
- [A practical example](#)
 - [The frequency filters](#)
 - [The spectrum analyzer](#)
- [Summary](#)
- [Miscellaneous](#)

Preface

This module is the first in a series of modules designed to teach you about Digital Signal Processing (DSP) using Java. The purpose of the miniseries is to present the concepts of DSP in a way that can be understood by persons having no prior DSP experience. However, some experience in Java programming would be useful. Whenever it is necessary for me to write a program to illustrate a point, I will write it in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

- [Figure 1](#). A sinusoidal function.
- [Figure 2](#). Complex harmonic motion.
- [Figure 3](#). Separate cosine and sine functions.
- [Figure 4](#). Sinusoid with frequency modification.
- [Figure 5](#). An approximate square waveform.
- [Figure 6](#). An improved approximate square waveform.
- [Figure 7](#). First five sinusoidal components of a square waveform.
- [Figure 8](#). A triangular waveform.

Preview

Many physical devices (*and electronic circuits as well*) exhibit a characteristic commonly referred to as *periodic motion* .

I will use the example of a pendulum to introduce the concepts of

- periodic motion,
- harmonic motion, and
- sinusoids.

I will introduce you to the concept of a *time series* .

I will introduce you to sine and cosine functions and the Java methods that can be used to calculate their values.

I will introduce you to the concepts of *period* and *frequency* for sinusoids.

I will introduce you to the concept of *radians* versus *cycles* .

I will introduce you to the concept of *decomposition* by decomposing a time series into a (*possibly very large*) set of sinusoids, each having its own frequency and amplitude. (*We will learn much more about this in a subsequent module when we discuss frequency spectral analysis.*)

I will introduce you to the concept of *composition*, where (*theoretically*) any time series can be created by adding together the correct (*possibly very large*) set of sinusoids, each having its own frequency and amplitude.

I will show you examples of using composition to create a square waveform and a triangular waveform.

I will identify some real-world examples of frequency filtering and real-time spectral analysis

Discussion and sample code

Periodic motion

The most difficult decision that I must make for this series is to decide where to begin. I need to begin at a sufficiently elementary level that you will understand everything that you read. So, before getting into the actual topic of digital signal processing, I'm going to take you back to some elementary physics and mathematical concepts and discuss *periodic motion*.

Many physical devices (*and electronic circuits as well*) exhibit a characteristic commonly referred to as periodic motion. This includes pendulums, acoustic speakers, springs, human vocal cords, etc.

Motion of a pendulum

For example, consider the pendulum on a grandfather clock. Once you start the pendulum swinging, it will swing for a long time (*actually, the spring in the clock gives it a little kick during each repetition, so it will continue to swing until the spring runs down*).

The most important characteristic of the motion of the pendulum is that every repetition takes almost exactly the same amount of time. In other words, the period of time during which the pendulum swings from one side to the other is very constant. That is the source of the term *periodic motion* .

A bag of sand

Even without the spring to help it along, a pendulum swinging in a low-friction environment will swing for a long time. Visualize a pendulum consisting of a bag of sand tied to the end of a long rope suspended from your ceiling (*sheltered from the wind*). Assume that the bag is filled with colored sand, and that it has a small hole in the bottom. Assume that you start it swinging in a back-and-forth motion (*no circular motion*).

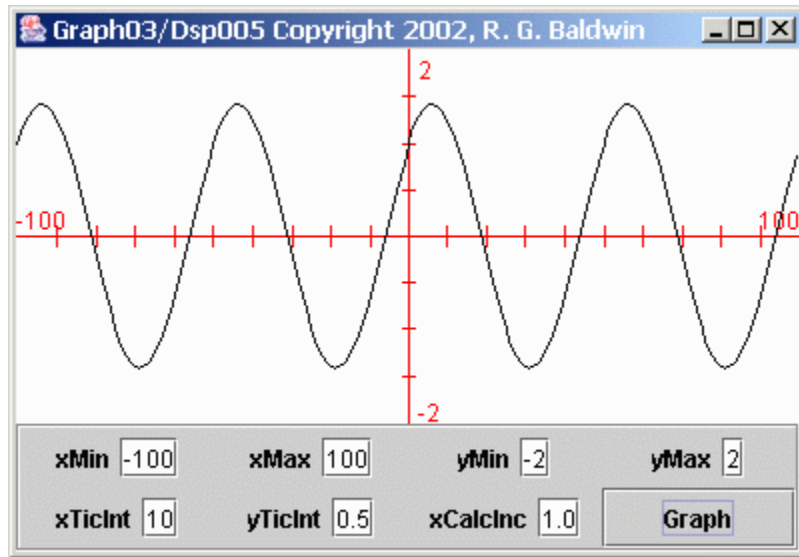
The sand traces out a pattern

As the bag of sand swings back and forth, a little bit of sand will leak out of the hole and land on the floor below. The sand will trace out a line, which is parallel to the direction of motion of the bag of sand.

Now assume that you carefully drag a carpet across the floor under the bag at a uniform rate, perpendicular to the direction of motion of the bag. This will cause the sand that leaks from the bag to trace out a zigzag pattern on the carpet very similar to the curved line shown in [Figure 1](#).

Figure 1. A sinusoidal function.

Figure 1. A sinusoidal function.



A sinusoidal function

The shape of the curve shown in [Figure 1](#) is often referred to as a *sinusoidal* function.

In this assumed scenario, the bag is swinging back and forth along the vertical axis in [Figure 1](#), and the carpet is being dragged along the horizontal axis. The motion of the bag causes the sand to be leaked in a back-and-forth zigzag pattern. The motion of the carpet causes that pattern to be elongated along the horizontal axis.

Note: A sinusoidal function:

[Figure 1](#) is a plot of the following Java expression:

```
Math.cos(2*pi*x/50) + Math.sin(2*pi*x/50)
```

Math in the above expression is the name of a Java library that provides methods for computing sine, cosine, and various other mathematical functions.

The asterisk (*) in the above expression means multiplication.

The reference to **pi** in the above expression is a reference to the mathematical constant with the same name.

The image in [Figure 1](#) was produced using a Java program named **Graph03** , which I will describe in a future module.

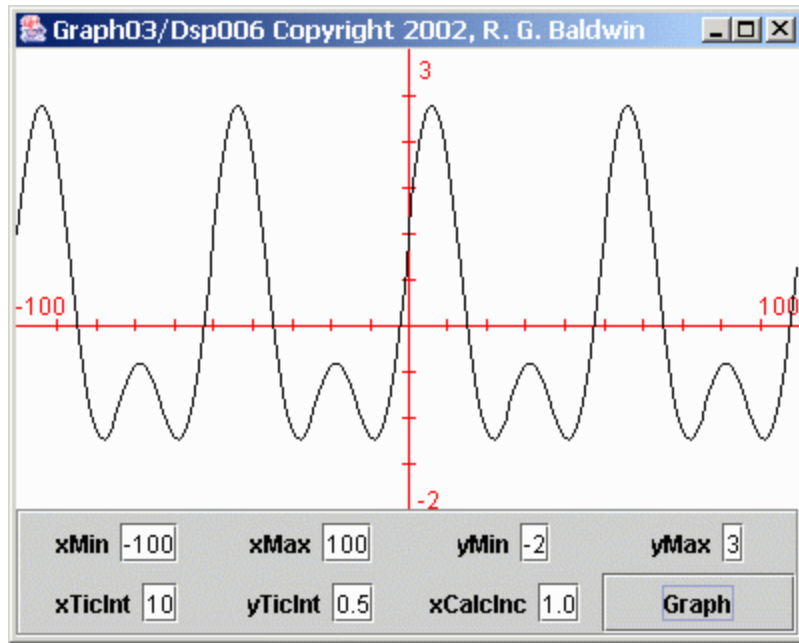
Harmonic motion

The kind of motion exhibited by a simple pendulum is often referred to as periodic motion in general, and simple harmonic motion in particular.

Other devices that exhibit periodic motion may exhibit more complex harmonic motion. For example, [Figure 2](#) illustrates a more complex form of harmonic motion.

Figure 2. Complex harmonic motion.

Figure 2. Complex harmonic motion.



As you can see, the positive excursions are greater than the negative excursions in [Figure 2](#). In addition, the negative excursions exhibit some ripple that is not exhibited by the positive excursions.

Note: A complex sinusoidal function:

[Figure 2](#) is a plot of the following Java expression:

```
Math.cos(2*pi*x/50)
+ Math.sin(2*pi*x/50)
+ Math.sin(2*pi*x/25);
```

Sinusoids

[Figure 1](#) shows a curve whose shape is commonly referred to as a sinusoid. Eliminating the syntax required by the Java programming language, the expression used to produce the data plotted in [Figure 1](#) was:

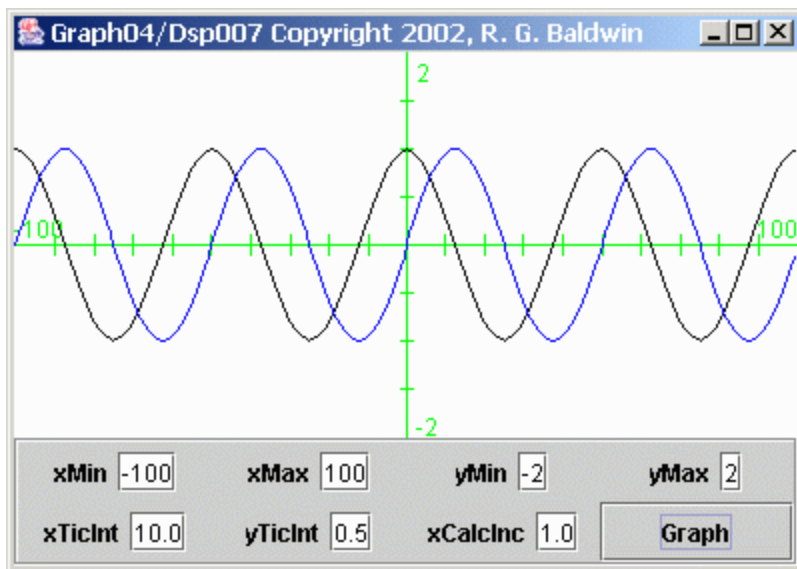
$$f(x) = \cos(2\pi x/50) + \sin(2\pi x/50)$$

As you can see, this function is composed of two components, one cosine component and one sine component.

Separate the cosine and sine components

[Figure 3](#) shows separate plots of the two components that were added together to produce the plot in [Figure 1](#).

Figure 3. Separate cosine and sine functions.



Plotting separate sine and cosine functions

The black curve in [Figure 3](#) shows the cosine function produced by evaluating and plotting the following equation:

$$f(x) = \cos(2\pi x/50)$$

The blue curve shows the sine function produced by evaluating and plotting the following equation:

$$g(x) = \sin(2\pi x/50)$$

The point-by-point sum of these two curves would produce the sinusoidal curve shown in [Figure 1](#).

Time varying functions

Many processes produce functions whose values vary over time. For example, the temperature in your office is a function that varies continuously with time.

(The temperature in your office is probably not a periodic function because the temperature probably doesn't repeat its values on any periodic basis).

Sampled data

If you were to measure and record the temperature in your office once each minute, you could plot the recorded values in the manner shown in [Figure 1](#) through [Figure 3](#). You could plot the temperature along the vertical axis against time (*or sample number*) along the horizontal axis.

A common name for sampled data of this type is *time series*. That name reflects the fact that your data is a recording of the temperature values for a

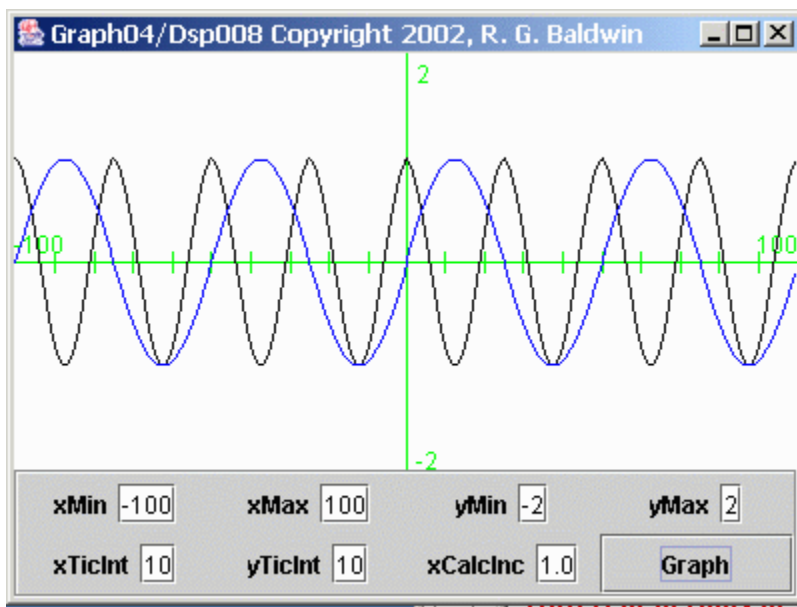
series of measurements that occur over time. (*I will have quite a lot more to say about time series in future modules.*)

The period of the sine and cosine functions

[Figure 4](#) is very similar to [Figure 3](#). However, there is one major difference. In [Figure 3](#), the repetition period of the sine and cosine functions is the same. In other words, the shape of the sine function is the same as the shape of the cosine function. They are simply shifted relative to one another along the horizontal axis.

Each of the curves in [Figure 3](#) has the same period, where the period is the horizontal distance from one peak to the next.

Figure 4. Sinusoid with frequency modification.



Different periods or frequencies

The black curve in [Figure 4](#) was produced by evaluating and plotting the following equation:

$$f(x) = \cos(2\pi x/25)$$

The blue curve in [Figure 4](#) was produced by evaluating and plotting the following equation:

$$g(x) = \sin(2\pi x/50)$$

Note that the periods (*frequencies*) for the black cosine curve and the blue sine curve in [Figure 4](#) are not the same. The period (*time interval between peaks*) for the black curve is one-half that of the blue curve. Stated differently, the frequency of the black curve is twice that of the blue curve.

(Frequency and period are reciprocals of one another. Period is a measure of the time required to complete one cycle of the function. Frequency is a measure of the number of cycles that are completed in a given amount of time, usually one second.)

Modified arguments

This difference exists because I modified the argument in the equation for the cosine function relative to the sine function. In particular, I halved the value in the denominator in the cosine argument relative to the denominator in the sine argument.

This caused the period of the cosine function to be only half as long as the period for the sine function. Stated differently, this caused the *frequency* of the cosine function to be twice the frequency of the sine function.

Why does pi appear in the arguments ?

Although it isn't obvious, for any given value of x , the arguments for the sine and cosine functions shown above are angular measurements in *radians*.

Most programming languages provide methods for computing the sine and the cosine values for an angle given in radians. That is true for Java, and it is also true for the plotting software used to produce these graphs. (*These graphs were produced using a Java program.*)

Radians versus cycles

One cycle constitutes 360 degrees.

(For example, this is the number of degrees that the big hand on a clock must traverse to begin at the 12 and make one complete cycle around the clock face and back to the 12.)

Beyond one cycle, the values of the sine and cosine functions repeat.

An angle of one radian is approximately equal to 57.3 degrees, and is exactly equal to 360 degrees divided by 2π . Therefore, in order to use a method that expects to receive an angle in radians, it is necessary to apply the correction factor of 2π as shown above to convert from cycles to radians.

The horizontal scale

The horizontal scale in [Figure 4](#) extends from minus 100 units to plus 100 units with a value of 0 in the center.

A tic mark appears every ten units. For x equal to 25, the argument for the cosine function equals 2π radians, or 360 degrees. A close examination of [Figure 4](#) shows that the cosine curve goes through one full cycle between an

x value of 0 and an x value of 25. Beyond that, the cosine function simply repeats.

Similarly, for x equal to 50, the argument for the sine function equals 2π radians, or 360 degrees. Again, a close examination of [Figure 4](#) shows that the sine curve goes through one full cycle between an x value of 0 and an x value of 50. Beyond that, the sine function simply repeats.

The argument to the sine and cosine functions

If you think of x as being a measurement of time in seconds, and $1/n$ being a measurement of frequency in cycles/second, the arguments to the sine and cosine functions can be viewed as:

$$2\pi(\text{radians/cycle}) * (x \text{ sec}) * (1/n) (\text{ cycle/sec})$$

If you cancel out like terms, this reduces to:

$$2\pi(\text{radians}) * (x) * (1/n)$$

Thus, with a fixed value of n , for each value of x , the argument represents an angle in radians, which is what is required for use with the functions of the Java **Math** library.

Where does $\sin(\text{arg})$ equal zero ?

The value of the sine of an angle goes through zero at every integer multiple of π radians. This explanation will probably make more sense if you refer back to [Figure 3](#).

The curve in [Figure 3](#) was calculated and plotted for n equal to 50. The sine curve has a zero crossing for every value of x such that x is a multiple of $n/2$, or 25.

Where are the peaks in the cosine function ?

Similarly, the peaks in the cosine curve in [Figure 3](#) occur for every value of x such that x is a multiple of $n/2$, or 25.

Composition and decomposition

In theory, it is possible to decompose any time series into a number (*quite possibly a very large number*) of sine and cosine functions each having its own amplitude and frequency. (*In a future module, we will learn how this is possible using a Fourier series or a Fourier transform.*)

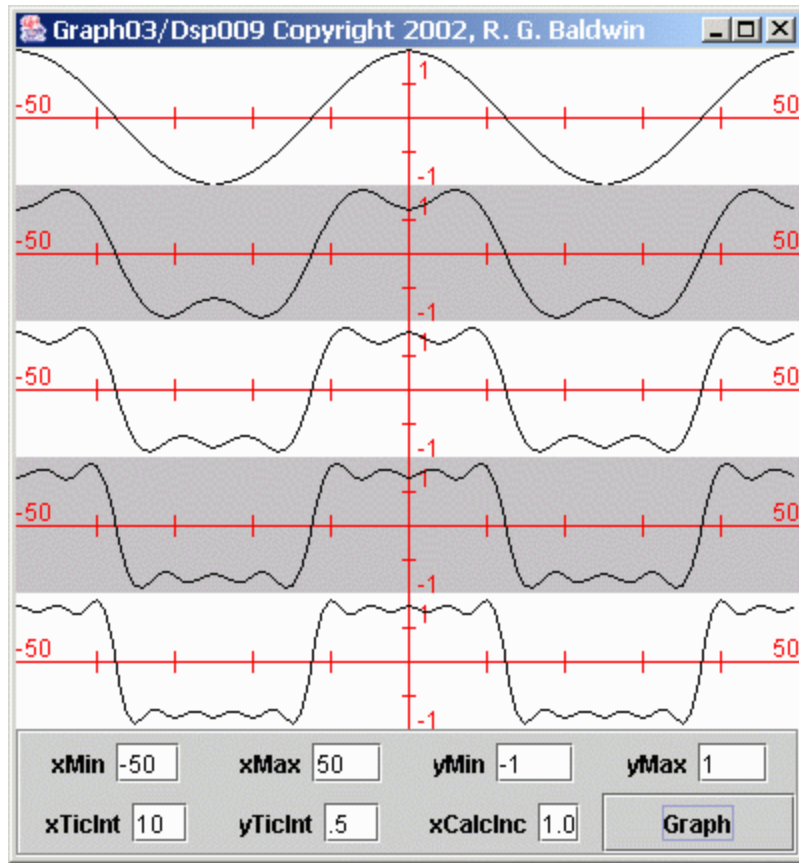
Conversely, it is theoretically possible to create any time series by adding together just the right combination of sine and cosine functions, each having its own amplitude and frequency.

An approximate square waveform

As an example of composition, suppose that I need to create a time series that approximates a square waveform, as shown at the bottom of [Figure 6](#).

Figure 5. An approximate square waveform.

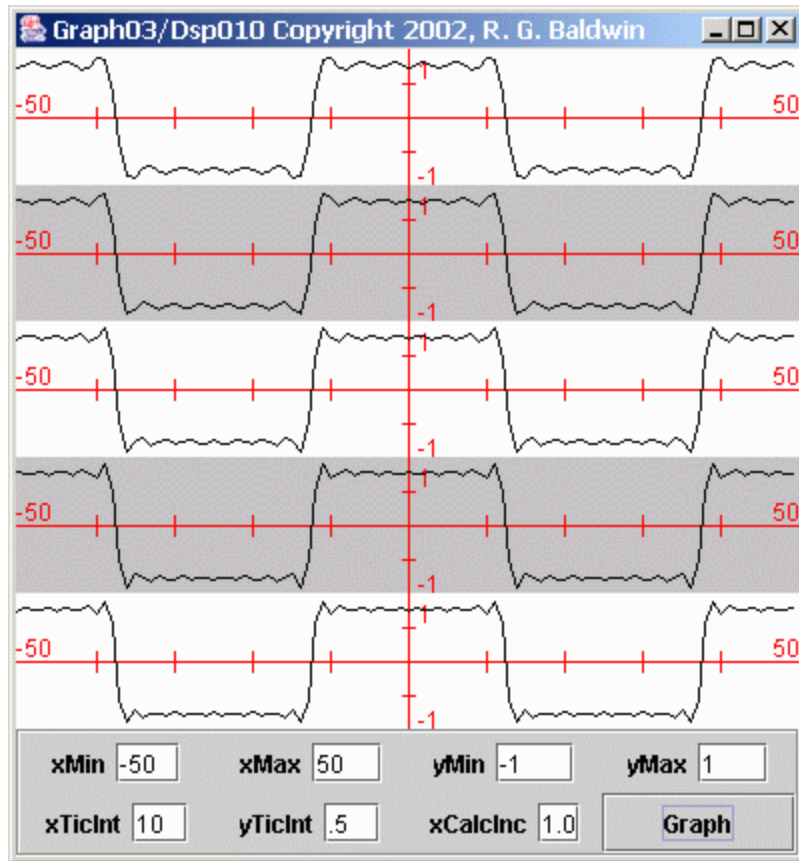
Figure 5. An approximate square waveform.



I can create such a waveform by adding together the correct combination of sinusoids, each having its own frequency and amplitude.

Figure 6. An improved approximate square waveform.

Figure 6. An improved approximate square waveform.



Successive approximations

The ten curves plotted in [Figure 5](#) and [Figure 6](#) show successive approximations to the creation of the desired square waveform. The bottom curve in [Figure 6](#) is a plot of the following sinusoidal expression containing the algebraic sum of ten sinusoidal terms.

$$\begin{aligned} &\cos(2\pi x/50) \\ &- \cos(2\pi x^3/50)/3 \\ &+ \cos(2\pi x^5/50)/5 \\ &- \cos(2\pi x^7/50)/7 \end{aligned}$$

$$\begin{aligned}
&+ \cos(2\pi x \cdot 9/50)/9 \\
&- \cos(2\pi x \cdot 11/50)/11 \\
&+ \cos(2\pi x \cdot 13/50)/13 \\
&- \cos(2\pi x \cdot 15/50)/15 \\
&+ \cos(2\pi x \cdot 17/50)/17 \\
&- \cos(2\pi x \cdot 19/50)/19
\end{aligned}$$

Each curve contains more sinusoidal terms

The top curve in [Figure 5](#) is a plot of only the first sinusoidal term shown above. It is a pure cosine curve.

Each successive plot, moving down the page in [Figure 5](#) and [Figure 6](#) adds another term to the expression being plotted, until all ten terms are included in the bottom curve in [Figure 6](#).

Reasonably good approximation

As you can see, the bottom curve in [Figure 6](#) is a reasonably good approximation to a square wave, but it is not perfect.

(A perfect square wave would have square corners, a flat top, no ripple, and perfectly vertical sides.)

Each term improves the approximation

If you start at the top of [Figure 5](#) and examine the successive curves, you will see that the approximation to a square wave improves as each new sinusoidal term is added.

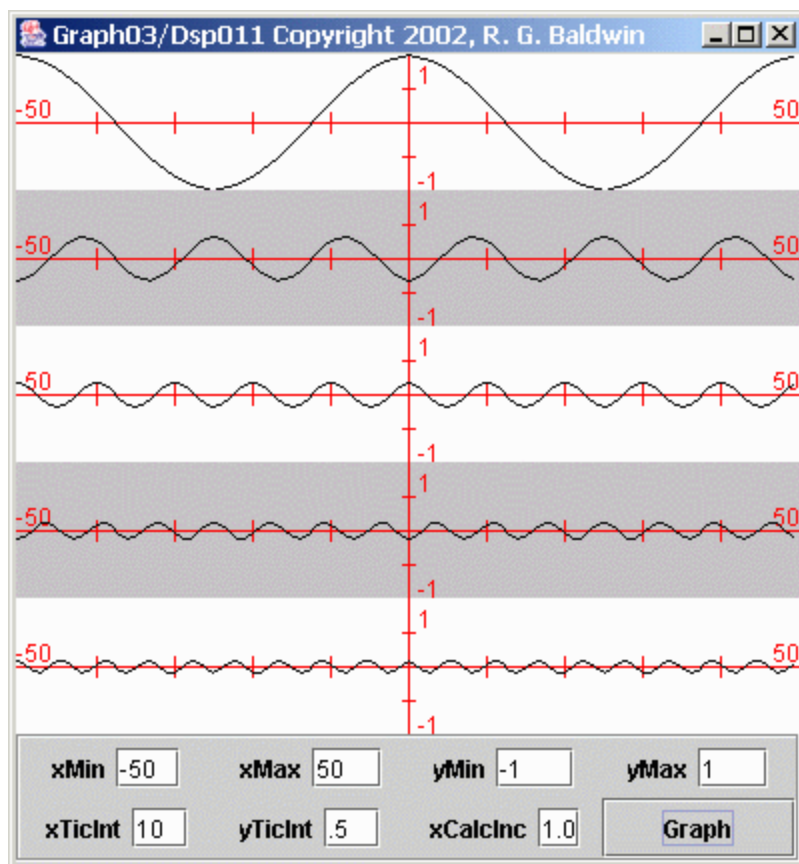
In theory, it would be possible to produce a perfect square wave in this fashion. Unfortunately, an infinite number of sinusoidal terms would be required to achieve the square corners, flat top, no ripples, and vertical sides

of the perfect square wave. In practice, we normally have to make do with something less than perfect.

The first five sinusoidal terms

[Figure 7](#) shows individual plots of the first five sinusoidal terms required to approximate the square wave.

Figure 7. First five sinusoidal components of a square waveform.



Each of the terms in the previous expression has an associated algebraic sign.

(You may have noticed that the sign applied to every other term in the expression is negative, causing every other term to plot upside down in [Figure 7](#).)

The bottom curve in [Figure 5](#) is the point-by-point sum of the five curves shown in [Figure 7](#).

A side-by-side comparison

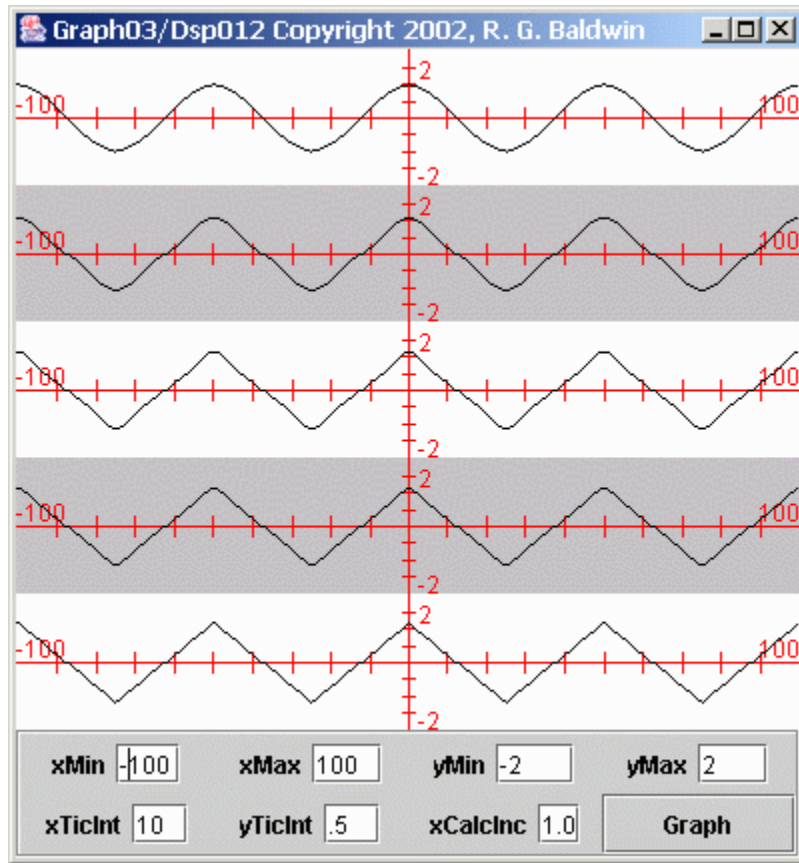
If you view [Figure 7](#) side-by-side with [Figure 5](#), you should be able to see how the sinusoidal terms add and subtract to produce the desired result. For example, the subtraction of the second sinusoidal term from the first sinusoidal term knocks the peaks off the first term and produces a noticeable shift from a cosine towards a square wave.

An approximate triangular waveform

As another example of composition, suppose that I need to create a time series that approximates a triangular waveform, as shown at the bottom of [Figure 8](#).

Figure 8. A triangular waveform.

Figure 8. A triangular waveform.



I can create such a waveform by adding together the right combination of sinusoids, each having its own amplitude and frequency.

Five sinusoids

The time series at the bottom of [Figure 8](#) was created by adding together five cosine waves, each having the amplitude and frequency values shown in the following expression:

$$\begin{aligned} f(x) = & \cos(2\pi x/50) \\ & + \cos(2\pi x^3/50)/9 \\ & + \cos(2\pi x^5/50)/25 \end{aligned}$$

$$+ \cos(2\pi x \cdot 7/50)/49$$

$$+ \cos(2\pi x \cdot 9/50)/81$$

Intermediate waveforms

The top waveform in [Figure 8](#) is a plot of the cosine curve created from the sinusoidal first term in the expression shown above.

The second waveform from the top in [Figure 8](#) is the sum of the first two terms in the above expression.

The third waveform is the sum of the first three terms. By this point, the plot has begun to resemble a triangular waveform in a significant way.

The fourth waveform is the sum of the first four terms, and the fifth waveform is the sum of all five terms.

By examining the waveforms from top to bottom in [Figure 8](#), you can see how the addition of each successive term causes the resulting waveform to more closely approximate the desired triangular shape.

Good result with only five terms

[Figure 8](#) shows that only five terms are required to produce a fairly good approximation to a triangular waveform.

A comparison of [Figure 8](#) with [Figure 5](#) shows that five terms are much more effective in approximating a triangular waveform than were the five terms in approximating a square waveform. The triangular waveform is easier to approximate because it doesn't have a flat top and vertical sides.

Other waveforms exhibit greater or lesser degrees of difficulty in creation through composition.

The individual terms

I'm not going to plot the individual sinusoidal terms in the triangular waveform. After the first couple of terms, they have such a small amplitude that it is difficult to see them.

So what ?

By now, you are may be saying "So what?" What in the world does DSP have to do with bags of sand with holes in the bottom? The answer is everything.

Almost everything that we will discuss in the area of DSP is based on the premise that every time series, whether generated by sand leaking from a bag onto a moving carpet, or acoustic waves generated by your favorite rock band, can be decomposed into a large (*possibly infinite*) number of sine and cosine waves, each having its own amplitude and frequency.

A practical example

You have probably seen, the kind of stereo music component commonly known as an equalizer. An equalizer typically has about a dozen adjacent slider switches that can be moved up and down to cause the music that you hear to be more pleasing. This is a crude form of a *frequency filter* .

Many equalizers also have a set of vertical display lights that dance up and down as your music is playing. This is a crude form of a *frequency spectrum analyzer* .

The frequency filters

The purpose of each slider is to attenuate or amplify a band of adjacent frequencies (*sine and cosine components, each having its own amplitude and frequency*), before they make their way to the output amplifier and impinge on the system speakers. Thus, while you don't have the ability to

attenuate or amplify each individual sine and cosine component, you do have the ability to attenuate or amplify them in groups.

In subsequent modules, we will learn how to use digital filters to attenuate or amplify the sine and cosine waves that make up a time series.

The spectrum analyzer

At an instant in time, the height of one of the vertical display lights is an indication of the combined power of the sine and cosine waves contained in a small band of adjacent frequencies.

In subsequent modules, you will learn how to use *Fourier analysis* to perform spectral analysis on time series.

Summary

Many physical devices (*and electronic circuits as well*) exhibit a characteristic commonly referred to as periodic motion.

I used the example of a pendulum to introduce the concepts of periodic motion, harmonic motion, and sinusoids.

I introduced you to the concept of a time series.

I introduced you to sine and cosine functions and the Java methods that can be used to calculate their values.

I told you that almost everything we will discuss in this series on DSP is based on the premise that every time series can be decomposed into a large number of sinusoids, each having its own amplitude and frequency.

I introduced you to the concepts of period and frequency for sinusoids.

I introduced you to the concept of radians versus cycles.

I introduced you to the concept of decomposing a time series into a (*possibly very large*) set of sinusoids, each having its own frequency and amplitude. I told you that we will learn more about this later when we discuss frequency spectrum analysis.

I introduced you to the concept of composition, where any time series can be created by adding together the correct (*possibly very large*) set of sinusoids, each having its own frequency and amplitude.

I showed you examples of using composition to create a square waveform and a triangular waveform.

I identified the frequency equalizer in your audio system as an example of frequency filtering.

I identified the frequency display that may appear on your frequency equalizer as an example of real-time spectrum analysis

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Dsp00100: Digital Signal Processing (DSP) in Java, Periodic Motion and Sinusoids
- File: Dsp00100.htm
- Published: 12/01/02

Baldwin kicks off a new miniseries on DSP. He discusses periodic motion and sinusoids. He introduces time series analysis, sine and cosine functions, and frequency decomposition. He discusses composition, and provides examples for square and triangular waveforms.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Dsp00104-Sampled Time Series

Baldwin explains the meaning of sampling, and identifies some of the problems that arise when sampling and processing analog signals. He explains the concept of the Nyquist folding frequency and illustrates the folding phenomena by plotting time series data as well as spectral data.

Revised: Fri Oct 16 23:11:24 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Preview](#)
- [Discussion](#)
 - [Sinusoids, time series, composition, and decomposition](#)
 - [The notion of sampling analog signals](#)
 - [What is meant by sampling?](#)
 - [The Nyquist folding frequency.](#)
 - [A few comments about sampling](#)
 - [What do we really have?](#)
 - [Sampled sinusoids](#)
 - [Estimating the values in between the samples](#)

- [A more common representation](#)
- [The most common representation](#)
- [What happens when the sampling frequency is reduced?](#)
 - [Some numbers](#)
 - [Comparison of frequencies with center frequency](#)
 - [Reduce the sampling frequency](#)
 - [A different approach](#)
 - [Introduce a sampling problem](#)
 - [Back to the case with no problems](#)
 - [Back to the case with the sampling problem](#)
- [The bottom line](#)
 - [Using an analog low-pass pre-filter](#)
 - [Digital re-sampling](#)
- [Summary](#)
- [Miscellaneous](#)

Preface

This is one in a series of modules designed to teach you about Digital Signal Processing (DSP) using Java. The purpose of the miniseries is to present the concepts of DSP in a way that can be understood by persons having no prior DSP experience. However, some experience in Java programming would be useful. Whenever it is necessary for me to write a program to illustrate a point, I will write it in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

- [Figure 1](#). Samples from five different sinusoids.
- [Figure 2](#). Rectangular representations of samples from five sinusoids.
- [Figure 3](#). Trapezoidal representations of samples from five sinusoids.
- [Figure 4](#). Most common representations of samples from five sinusoids.
- [Figure 5](#). Five sampled sinusoids
- [Figure 6](#). Result of re-sampling the five sinusoids.
- [Figure 7](#). Spectral analyses of five sinusoids with no sampling problems.
- [Figure 8](#). Spectral analyses of five sinusoids with sampling problem.
- [Figure 9](#). Spectral analyses of five sinusoids with no sampling problems.
- [Figure 10](#). Spectral analyses of five sinusoids with sampling problem

Preview

I will explain the meaning of sampling, and will explain some of the problems that arise when sampling and processing analog signals. Those problems generally relate to the relationship between the sampling frequency and the high-frequency components contained in the analog signal.

I will explain the concept of the *Nyquist folding frequency*, which is half the sampling frequency (*more commonly called the sampling rate*).

I will illustrate the frequency folding phenomena by plotting sampled time series data as well as spectral data.

Discussion

Sinusoids, time series, composition, and decomposition

I introduced you to sinusoids and sampled time series in an [earlier module](#). I taught you about sine and cosine functions, and the Java methods used to

calculate their values. I also introduced the concepts of *period* and *frequency* for sinusoids.

While introducing *decomposition* , I told you that almost everything we will discuss in this series on DSP is based on the premise that every time series can be decomposed into a large number of sinusoids, each having its own amplitude and frequency.

I also introduced the concept of *composition* , where any time series can be created by adding together the correct set of sinusoids, each having its own amplitude and frequency.

The notion of sampling analog signals

While signal processing can be accomplished in a variety of ways, including analog processors, digital processors, and optical processors, DSP is based on the notion that signals in nature can be sampled and converted into a series of numbers. The numbers can be fed into some sort of digital device, which can process the numbers to achieve some desired objective.

What is meant by sampling ?

To sample a signal means to measure and record its amplitude at a series of points in time. For example, you might record the temperature in your office every ten minutes for twenty-four hours. In this case, the actual temperature in your office would be the analog signal. The 144 temperature values that you record would be a sampled time series intended to represent that analog signal.

Uniform sampling is most common

Although uniform sampling is not strictly necessary, in DSP, the most common practice is to sample the signal at uniform intervals of time, (*such*

as once every ten minutes, once per second, or one-thousand times per second). This results in a uniform sampling frequency (sampling rate) .

(Most of the discussions in this series of tutorials on DSP will assume a uniform sampling frequency.)

Some problems arise

While sampled data can be used to simulate most of the signal-processing capabilities available with analog devices, the process of sampling does introduce some complications that must be dealt with. For the most part, these complications have to do with the relationship between the sampling frequency (*in samples per second*) and the highest frequency component contained in the signal (*in cycles per second*).

Stated simply, if the analog signal contains any sinusoidal components whose frequency is greater than half the sampling frequency, then those components will appear in the sampled time series at a different frequency. This can result in a variety of problems.

Reconstruction of the analog signal

Theoretically, if the sampling frequency is twice the highest frequency component contained in the analog signal, then the samples can be used in conjunction with an analog filter to reconstruct the original analog signal.

(However, this requires the construction of a perfect analog filter. In practice, the sampling frequency needs to be perhaps five to ten times the highest frequency component in the analog signal to make it practical to do a good job of reconstructing the analog signal from the samples.)

Reconstruction is not always required

Once the signal has been sampled and converted to digital form, there is often no interest in reconstructing the analog signal from the samples. While this eliminates the difficulty of reconstruction, it doesn't eliminate the potential problems caused by having the sampling frequency be less than twice the highest frequency component in the signal.

The Nyquist folding frequency

If the analog signal contains frequency components that are greater than half the sampling frequency, those components will appear to be at a different frequency in the sampled data.

The frequency that is equal to half the sampling frequency is often referred to as the *Nyquist folding frequency*, or simply the *folding frequency*. The folding frequency is half the sampling frequency. I will provide examples later to illustrate where this frequency gets its name.

A brief description

If a frequency component in the analog signal is less than the sampling frequency, but exceeds the folding frequency by an amount d , it will appear in the sampled data at a frequency that is the folding frequency minus d .

In other words, the entire frequency spectrum appears to fold around the *folding frequency* such that all frequency components that are above the folding frequency fold down to a similar position on the lower side of the folding frequency. Those frequency components above the folding frequency produce a mirror image below the folding frequency.

(If a frequency component in the analog signal is greater than the sampling frequency, folding still occurs, but in a more complicated way.)

Some specific numbers

Some specific numbers may make this easier to understand. Assume that the sampling frequency is 2000 samples per second, giving a folding frequency of 1000 cycles per second.

If an analog signal contains a frequency component at 1100 cycles per second, it will fold down and appear at 900 cycles per second in the sampled signal.

A frequency component at 1600 cycles per second in the analog signal will fold down and appear at 400 cycles per second in the sampled signal.

A frequency component at 2000 cycles per second (*the sampling frequency*) will fold down and appear at zero frequency in the sampled signal.

A few comments about sampling

The folding behavior is fairly easy to illustrate graphically, and I will do that shortly. Before doing that, however, I need to make a few comments about what it really means to sample an analog signal.

What do we really have ?

First we need to think about what we really have when we have a sampled time series. All that we really have is a set of values taken at specific times.

In reality, we know nothing about the values that actually existed for the analog signal in-between the samples.

For example, in the temperature experiment described earlier, when we record the temperature once every ten minutes, we can't really say what values we would have recorded if we had recorded the temperature once every five minutes instead. Therefore, we sometimes find ourselves estimating what the values are between the recorded samples.

Sampled sinusoids

Consider the five plots shown in [Figure 1](#).

Figure 1. Samples from five different sinusoids.

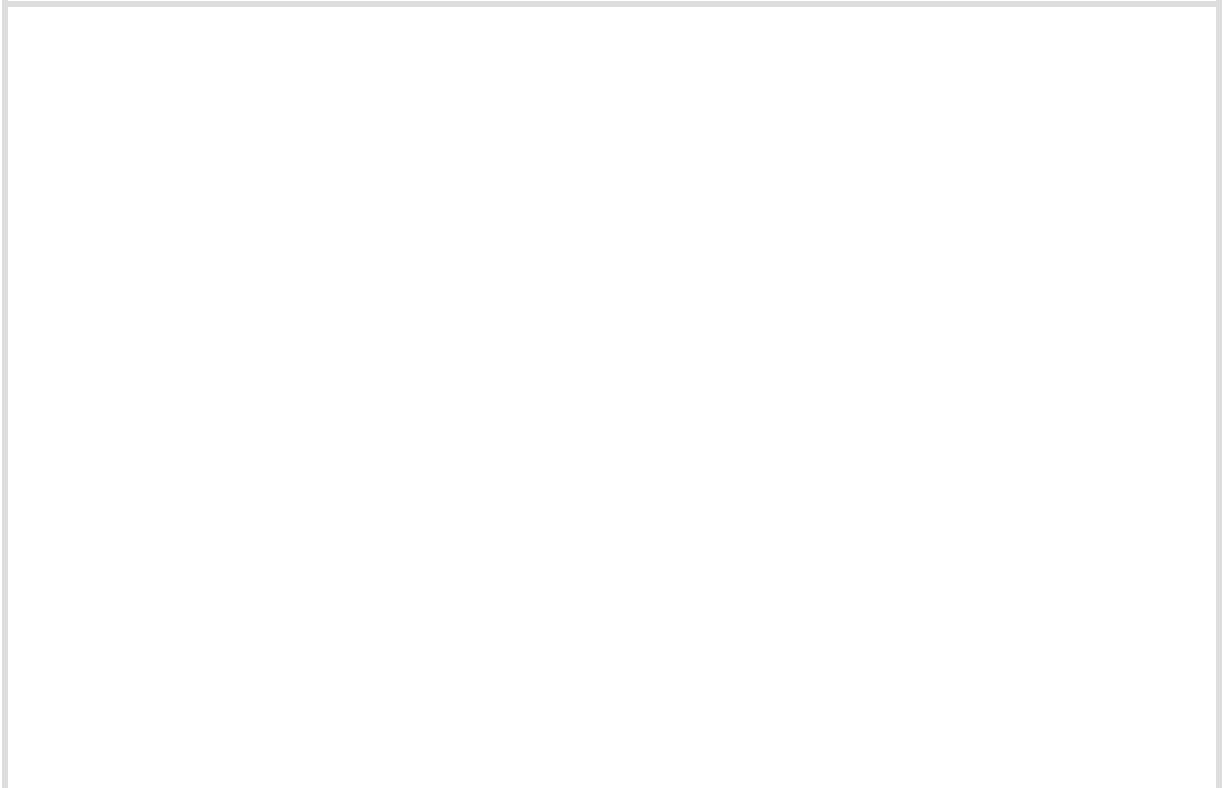
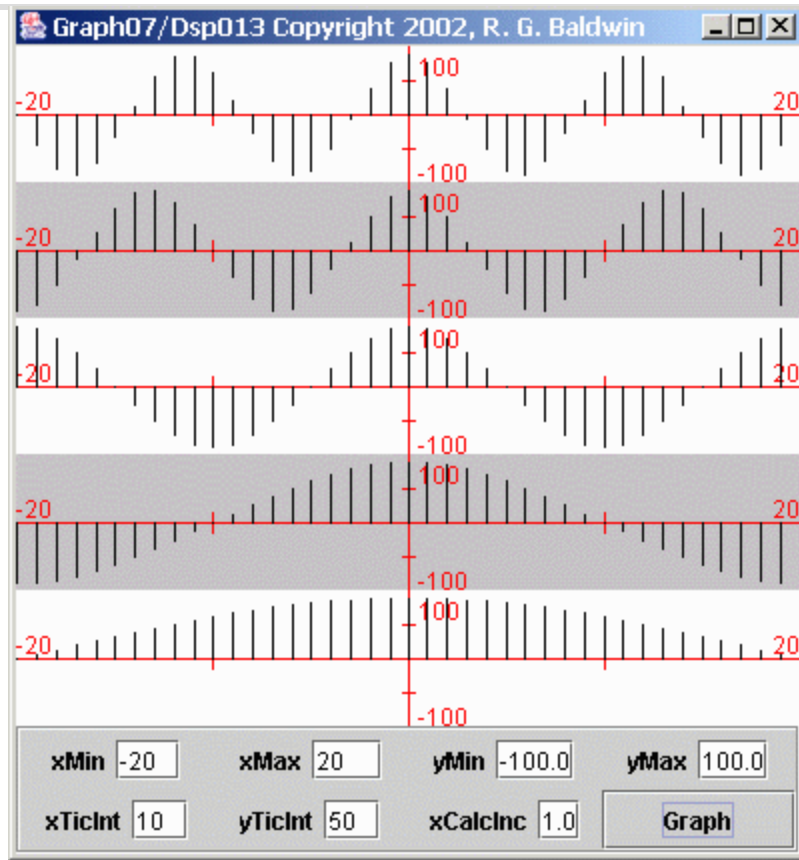


Figure 1. Samples from five different sinusoids.



[Figure 1](#) shows the values for samples taken from five different sinusoids (the height of each vertical bar represents the value of a sample).

All five sinusoids were sampled at the same sampling frequency. The sinusoid in the center was sampled twenty times per cycle (*not necessarily twenty times per second*).

The two sinusoids above the center had higher frequencies than the sinusoid in the center, with the sinusoid at the top having the highest frequency. For a fixed sampling frequency, the sinusoids above the center had fewer samples per cycle than the sinusoid in the center. The sinusoid at the top had the fewest number of samples per cycle.

The two sinusoids below the center had lower frequencies, than the sinusoid in the center, with the sinusoid at the bottom having the lowest frequency.

The two sinusoids below the center had more samples per cycle than the sinusoid in the center. The sinusoid at the bottom had the most samples per cycle.

The number of samples per cycle is important

In the final analysis, what really counts is not the number of samples per second of the sampling frequency, or the number of cycles per second of the signal frequency. What really counts is the number of ***samples per cycle*** of the highest frequency component. This value is established by the combination of the signal frequency and the sampling frequency.

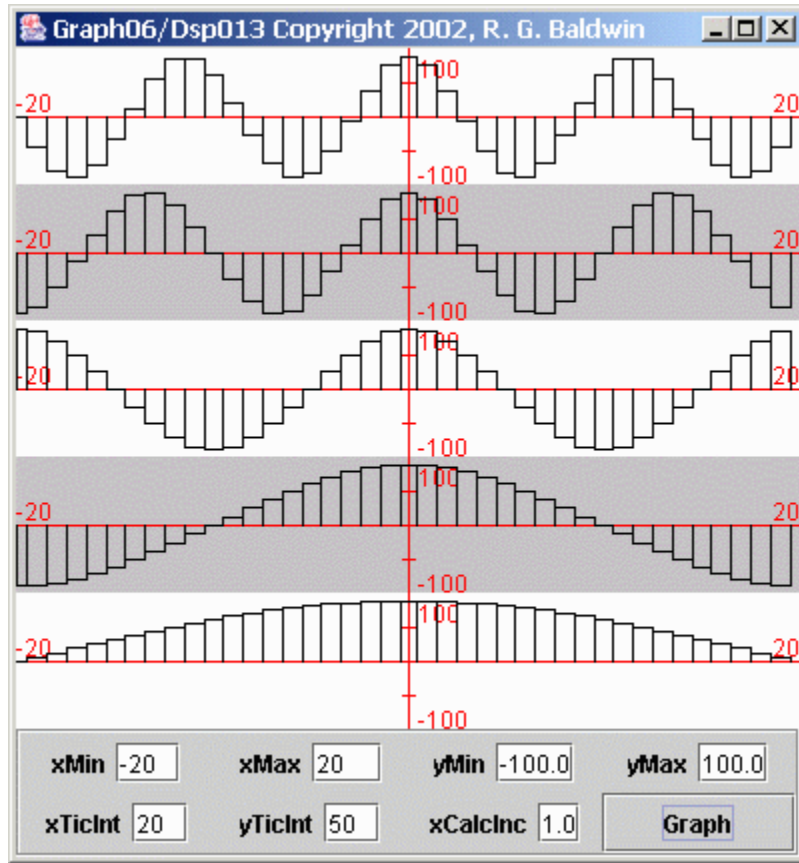
The values between the samples

Because the plots in [Figure 1](#) are pure sinusoids, I can mathematically determine the values between the samples. However, if there had been the slightest amount of random noise superimposed on the sinusoids, (*which is the more realistic situation*), I would have no way of knowing the values between the samples. Thus, all of the information that I have about these five signals is contained in the heights of the vertical bars shown in [Figure 1](#).

Estimating the values in between the samples

As mentioned earlier, we often find ourselves estimating the values in between the samples. One way to do this is shown in [Figure 2](#), which shows a different graphical treatment for the same five sinusoids.

Figure 2. Rectangular representations of samples from five sinusoids.

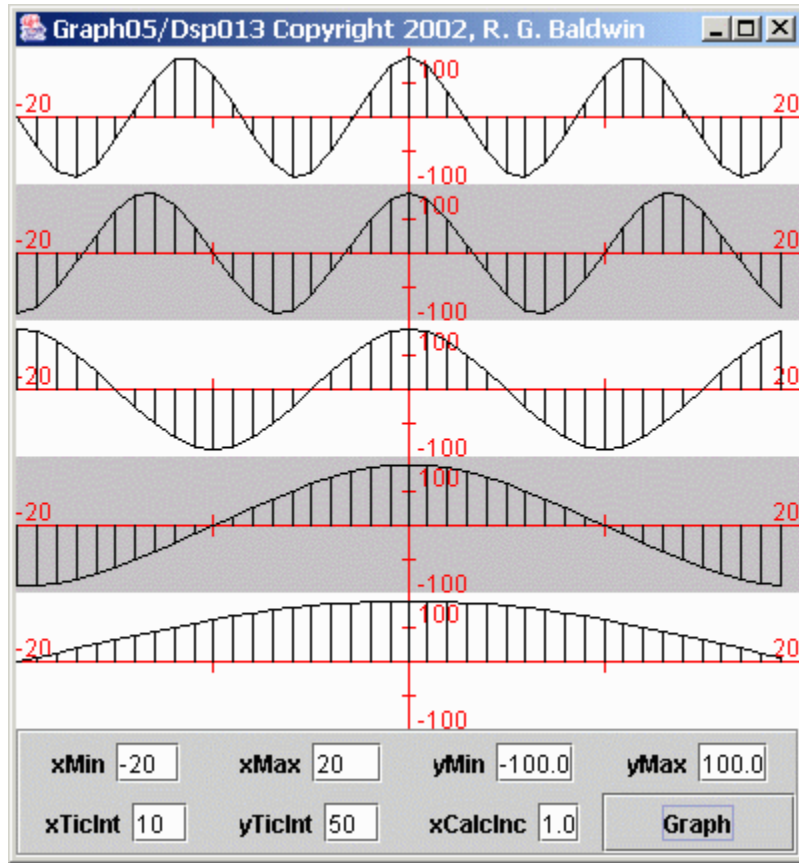


[Figure 2](#) represents each of the sample values as a rectangle. In effect, this treatment estimates that there is no change in the value of the analog signal for half a sample interval after the sample is taken. Then the value of the analog signal jumps to the value of the next sample.

A more common representation

Now consider the graphical treatment for the same five sinusoids shown in [Figure 3](#).

Figure 3. Trapezoidal representations of samples from five sinusoids.

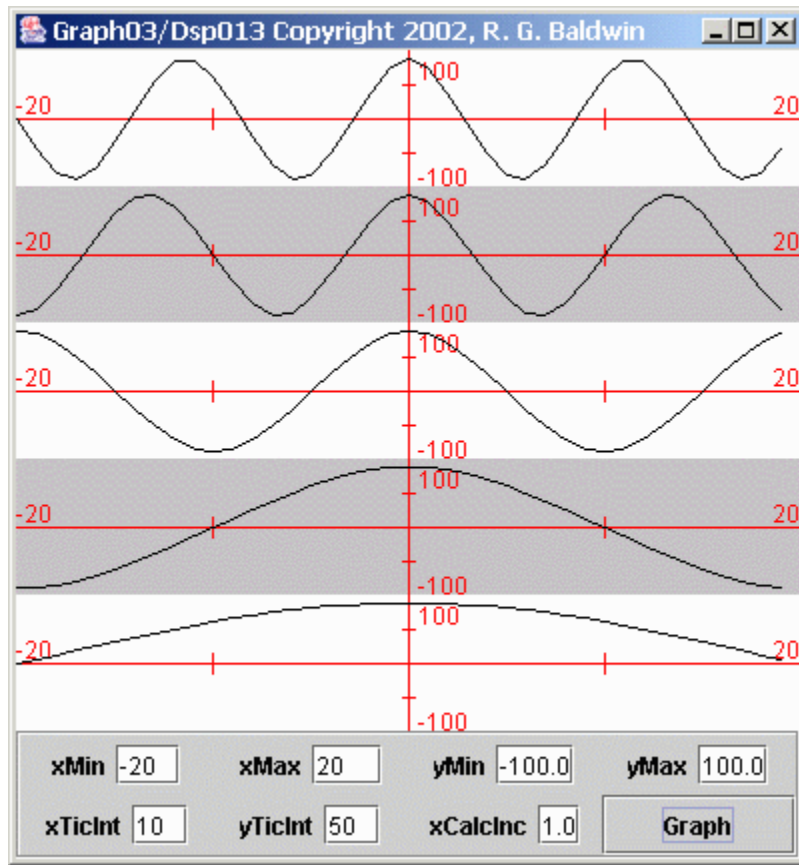


[Figure 3](#) shows a more common representation of the data. [Figure 3](#) treats each sample as a trapezoid consisting of a rectangle and a right triangle. The triangle sets atop the rectangle and connects each sample value to the next with a straight line.

The most common representation

Now consider the most common representation of the sampled data, as shown in [Figure 4](#).

Figure 4. Most common representations of samples from five sinusoids.



[Figure 4](#) shows the most common representation of the sampled data. [Figure 4](#) is the same as [Figure 3](#) except that the vertical lines that identify the sides of the trapezoids have been omitted. In [Figure 4](#), each sample value is connected to the next sample value with a straight-line segment.

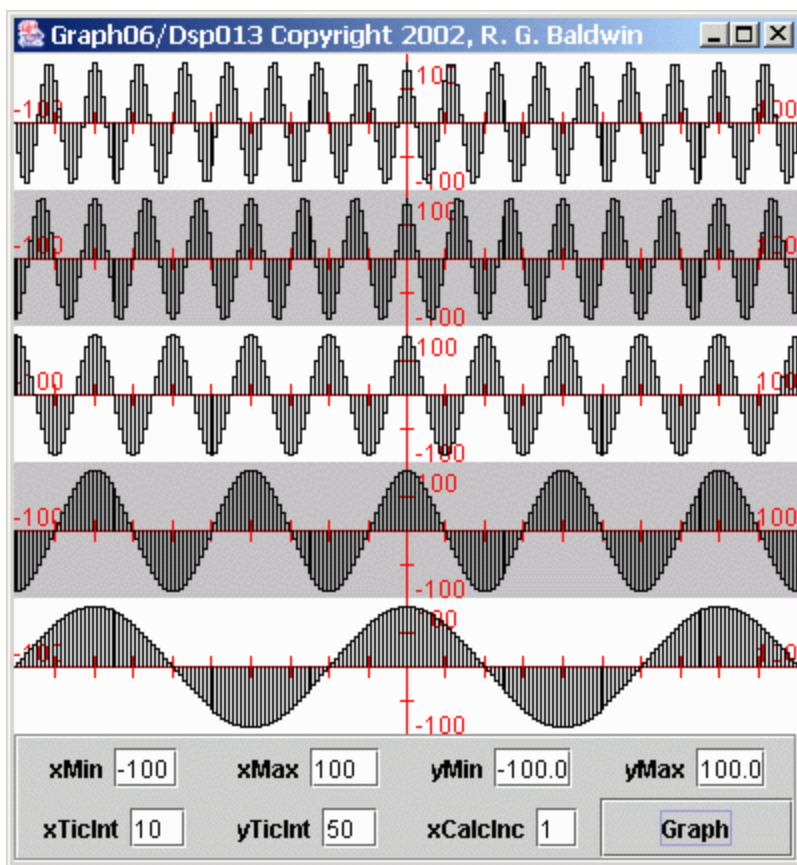
What happens when the sampling frequency is reduced ?

As you can see in these figures, regardless of which graphical treatment you use, the sampling frequency relative to the signal frequency is sufficiently high to present a respectable view of the sinusoidal signals. Now I'm going

to show you what happens when the sampling frequency is reduced without changing the frequency of the sinusoids.

[Figure 5](#) shows the same five sinusoids as above, except that they are plotted across a longer period of time. (The presentation in [Figure 5](#) treats each sample as a rectangle.)

Figure 5. Five sampled sinusoids



In particular, you should note the obvious frequency difference between the top sinusoid and the bottom sinusoid. Also note the frequency difference

between the two sinusoids immediately above and immediately below the center sinusoid.

Some numbers

Let's put some numbers to the frequencies involved. If we consider the sampling frequency to be 20 samples per second, then the center sinusoid has a frequency of one cycle per second, with 20 samples per cycle. On that basis, the frequencies of the sinusoids from top to bottom are as shown below:

1.75 cycles per second
1.50 cycles per second
1.00 cycles per second
0.50 cycles per second
0.25 cycles per second

Comparison of frequencies with center frequency

The most important thing to note about these frequency values is how the four outer frequencies relate to the center frequency. The top and bottom frequency values differ from the center frequency by 0.75 cycles per second. In other words, the frequency of the top sinusoid is 0.75 cycles per second above the frequency of the center sinusoid, and the frequency of the bottom sinusoid is 0.75 cycles per second below the frequency of the center sinusoid.

Similarly, the second and fourth frequency values differ from the center frequency by 0.50 cycles per second. Again, one is above and the other is below.

Reduce the sampling frequency

What I am going to do now is to recalculate and re-plot the values for each sinusoid at a sampling frequency of two samples per second instead of 20 samples per second. This will place the frequency of the center sinusoid exactly at the folding frequency of one cycle per second. More importantly, this will place the frequencies of the top two sinusoids above the folding frequency.

Re-plot the sampled data

I will re-plot the data for each sinusoid across the same period of time as in [Figure 5](#). The results are shown in [Figure 6](#). It would probably be useful for you to view [Figure 5](#) and [Figure 6](#) side-by-side in separate browser windows.

Figure 6. Result of re-sampling the five sinusoids.

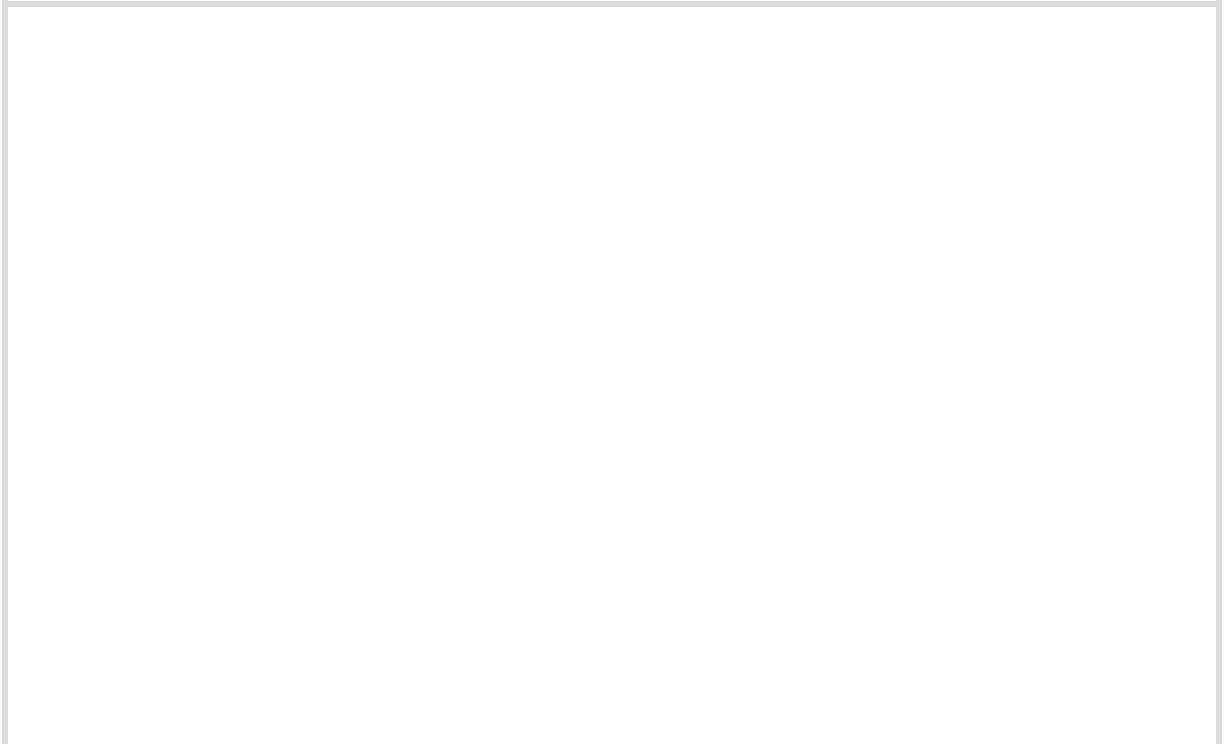
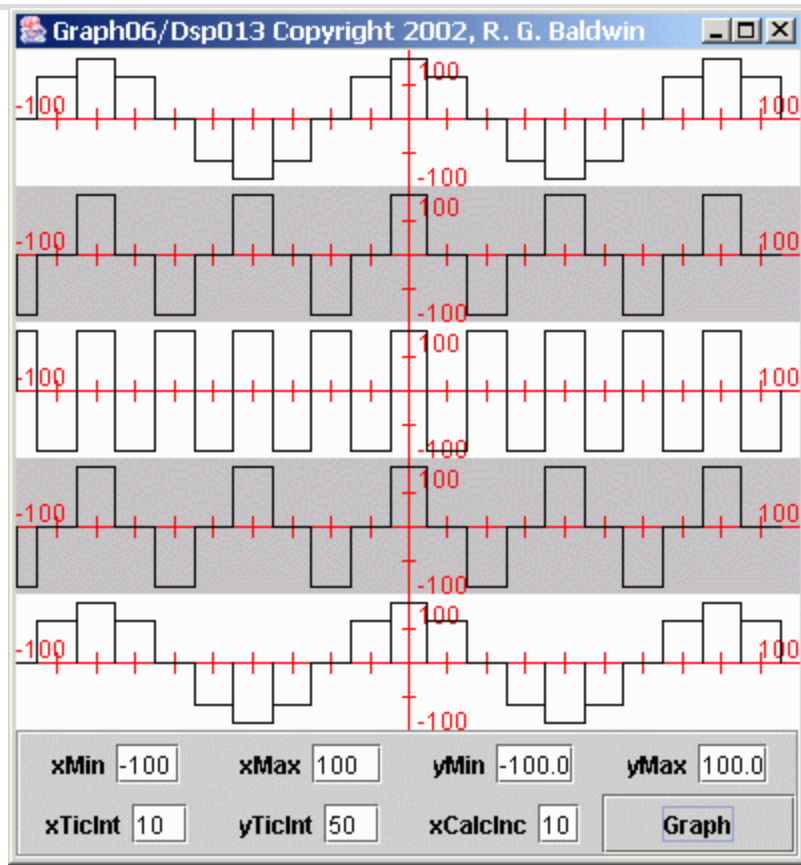


Figure 6. Result of re-sampling the five sinusoids.



There are several important things to note about [Figure 6](#).

Center plot no longer resembles a sinusoid

First, you will probably notice that the plot for the center sinusoid no longer looks much like a sinusoid. Rather, it looks like a square wave. This is the result of having exactly two samples per cycle of the sinusoid. One sample is taken from the positive lobe of the sinusoid, and the next sample is taken from the negative lobe of the sinusoid. This pattern repeats, producing something that looks like a square wave. (*A different graphical treatment would make it look like a triangular wave.*)

The top two sinusoids

More important, however, is to note what has happened to the top two sinusoids. Because the frequencies of the top two sinusoids are above the folding frequency, they no longer have a minimum of two samples per cycle. Thus, the apparent frequency of these two sinusoids has folded around the folding frequency and appears as a lower frequency.

Top sinusoids match bottom sinusoids

In fact, the plot of the top sinusoid now looks exactly like the plot of the bottom sinusoid at a frequency of 0.25 cycles per second. This means that the energy in the top sinusoid at 1.75 cycles per second has folded into a new frequency of 0.25 cycles per second.

The plot of the sinusoid immediately above the center looks exactly like the plot of the sinusoid immediately below the center at a frequency of 0.50 cycles per second. This means that the energy in that sinusoid at 1.5 cycles per second has folded into a new frequency of 0.5 cycles per second.

Bottom three plots are correct

The plots of the center sinusoid and the two sinusoids below the center are still correct (*although not very well sampled*). However, the frequency information embodied in the top two sinusoids has been lost. The top two sinusoids appear to be at a different frequency than the actual frequency of the corresponding analog signals. The fact that the frequencies of these two sinusoids were originally 1.75 and 1.50 cycles per second is now lost in the sampled data.

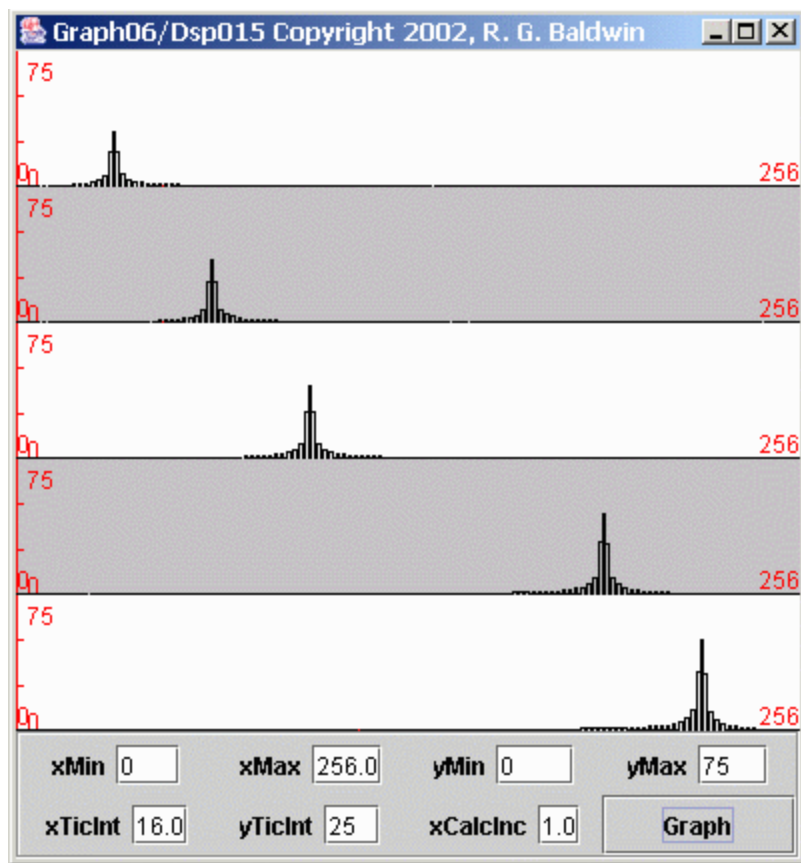
A different approach

Now I'm going to illustrate the same folding phenomena from a different perspective using spectral analyses. First I will show you a case having no

sampling problems. Then I will introduce a sampling problem and show you the impact that the problem has on the final results.

[Figure 7](#) shows the result of performing spectral analyses on five different sinusoids (*not the same five as in the previous discussion*). Each plot in [Figure 7](#) shows the spectrum of a different sinusoid. The spectrum is computed and displayed from zero frequency on the left to the folding frequency on the right.

Figure 7. Spectral analyses of five sinusoids with no sampling problems.



Sampling frequency was four samples per second

The sampling frequency for the data in [Figure 7](#) was four samples per second, giving a folding frequency of two cycles per second. Thus, the horizontal scale on each plot represents the frequencies from zero on the left to two cycles per second on the right.

The five sinusoids

Starting at the top, each of the five plots represents the frequency spectrum of a sinusoid having the amplitude and frequency shown in the following table.

Note: Amplitudes and frequencies of sinusoids:

Plot	Amplitude	Frequency
1	60	0.25 cycles per second
2	70	0.50 cycles per second
3	80	0.75 cycles per second
4	90	1.50 cycles per second
5	100	1.75 cycles per second

The heights of the spectral peaks

The height of each spectral peak in [Figure 7](#) is consistent with the amplitude of the corresponding sinusoid given in the table.

The locations of the spectral peaks

The spectral peaks in [Figure 7](#) appear where you would expect to see them. For example, the location of the peak in the first plot corresponds to a frequency of 0.25 cycles per second within a total frequency range extending from zero to two cycles per second. This matches the information given in the above table for the first sinusoid.

The location of the spectral peak in the fifth plot corresponds to a frequency of 1.75 cycles per second within a total frequency range extending from zero to two cycles per second. This matches the information given in the above table for the fifth sinusoid.

The location of the peak in each of the three plots between the first and the last are correct for the frequency of the sinusoid involved.

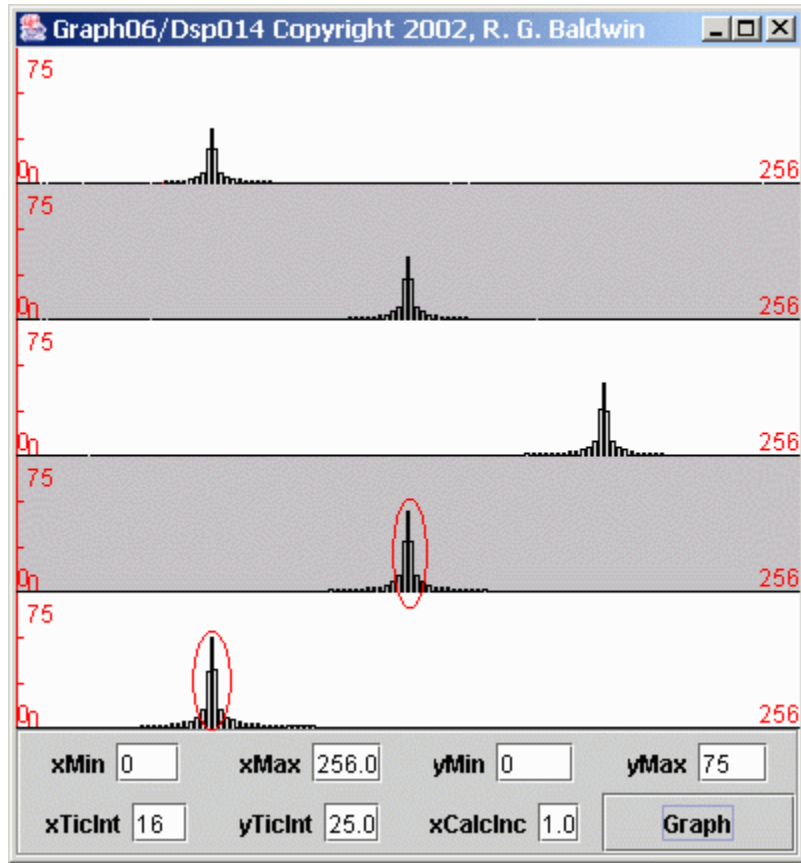
Introduce a sampling problem

Now I will introduce a sampling problem by keeping the frequencies of the sinusoids the same and reducing the sampling frequency from four samples per second to two samples per second.

The result of this change is shown in [Figure 8](#).

Figure 8. Spectral analyses of five sinusoids with sampling problem.

Figure 8. Spectral analyses of five sinusoids with sampling problem.



As before, each of the plots in [Figure 8](#) shows the frequency spectrum of an individual sinusoid. The spectrum is plotted from zero frequency on the left to the folding frequency on the right.

Sampling frequency was two samples per second

In this case, the sampling frequency was two samples per second, giving a folding frequency of one cycle per second. Therefore, the horizontal scale on each plot represents the frequencies from zero on the left to one cycle per second on the right.

The heights of the spectral peaks

Once again, the height of each spectral peak is consistent with the amplitude of the sinusoid.

The locations of the spectral peaks

As before, the spectral peaks in the first three plots appear where you would expect to see them. The peak in the first plot is about twenty-five percent of the way across the total spectrum, corresponding to 0.25 cycles per second.

The spectral peak in the second plot is at the center, corresponding to 0.5 cycles per second. The third peak is in the correct location for 0.75 cycles per second.

A problem with the location of two spectral peaks

However, a problem exists with the spectral peaks in the last two plots.

(I marked the two problem peaks with a red oval to make it obvious which ones I am talking about. You may find it helpful to compare [Figure 8](#) side-by-side with [Figure 7](#).)

The spectral peak in the fourth plot also appears about midway between zero and one cycle per second. This indicates that the corresponding sinusoid had a frequency of 0.5 cycles per second.

However, the frequency of the sinusoid for the fourth plot was 1.50 cycles per second, not 0.5 cycles per second as indicated. Thus, that spectral peak should have been off the scale on the right-hand side of the plot.

The folding frequency

Recall, however, that the right edge of the plot is the folding frequency. Therefore, any spectral components that should appear to the right of the folding frequency fold around and appear to the left of the folding frequency. Therefore, the spectral peak in the fourth plot, which should appear at 0.50 cycles per second above the folding frequency, appears instead at 0.50 cycles per second below the folding frequency.

The peak in the fifth plot

Similarly, the frequency of the sinusoid for the fifth plot was 1.75 cycles per second. The peak for this sinusoid should have appeared 0.75 cycles per second above the folding frequency, but appeared instead 0.75 cycles per second below the folding frequency. In other words, the spectrum folded around the folding frequency so that this peak appeared below the folding frequency.

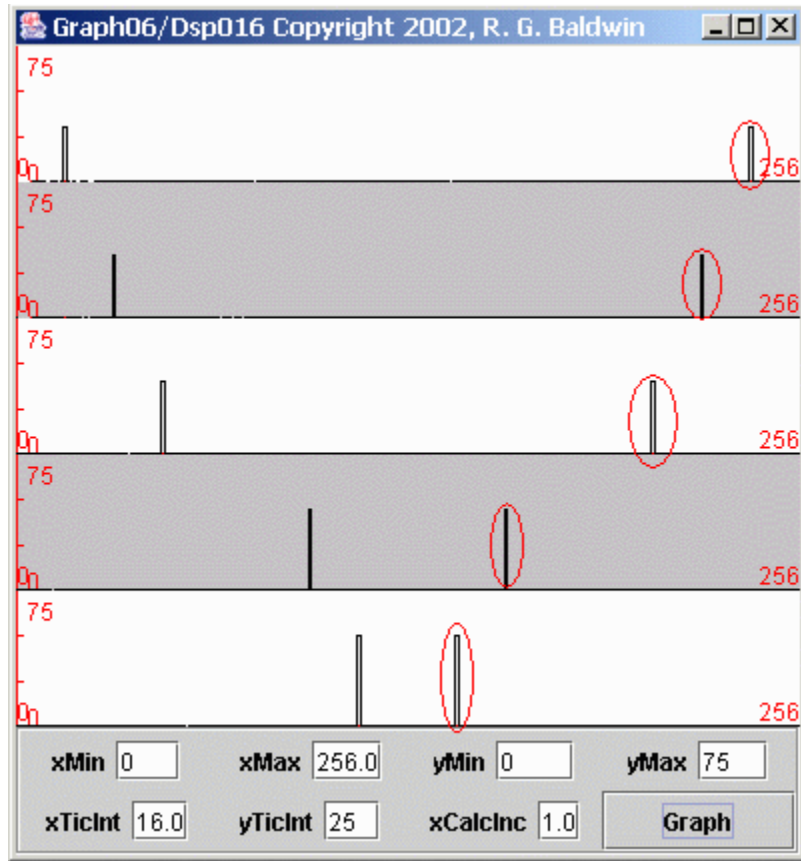
I am going to show you two more views of the spectra of these sinusoids to help you better understand the folding phenomena.

Back to the case with no problems

Let's go back and examine another view of the case that has no sampling problems. This view is shown in [Figure 9](#).

Figure 9. Spectral analyses of five sinusoids with no sampling problems.

Figure 9. Spectral analyses of five sinusoids with no sampling problems.



Sampled at four samples per second

This is the case where all five sinusoids are sampled at a sampling frequency of four samples per second, resulting in a folding frequency of two cycles per second. If you compare [Figure 9](#) with [Figure 7](#), you will see that the left half of [Figure 9](#) is very similar to [Figure 7](#).

[Figure 9](#) shows twice the frequency range

In [Figure 7](#), the spectral data was computed and displayed from zero frequency on the left to the folding frequency (*two cycles per second*) on the right. In [Figure 9](#), the spectral data was computed and displayed from zero frequency on the left to the sampling frequency (*four cycles per second*) on the right.

Thus, the total frequency range for [Figure 9](#) is twice the frequency range for [Figure 7](#).

Folding frequency at the center

In [Figure 9](#), the folding frequency is exactly in the center of each plot. In other words, the center of the plots in [Figure 9](#) corresponds to the right edge of the plots in [Figure 7](#). Everything to the left of center in [Figure 9](#) corresponds to the plots in [Figure 7](#). The material to the right of center in [Figure 9](#) was not shown in [Figure 7](#).

Why is it called the folding frequency?

Hopefully the display in [Figure 9](#) will explain why the frequency that is half the sampling frequency is called the folding frequency. The computed spectrum folds around that frequency. Everything to the right of the folding frequency is a mirror image of everything to the left of the folding frequency.

Peaks below folding frequency are valid

All the peaks to the left of center in [Figure 9](#) are valid spectral peaks associated with the corresponding sinusoids. However, all the peaks to the right of center, which I marked with red ovals, are artifacts of the sampling process. Those peaks do not exist in the true spectrum of the original raw data. They were created by the sampling process.

Normally don't compute the mirror image

Normally we don't worry about this mirror image above the folding frequency when doing spectral analyses. We know it is there and we simply ignore it.

In fact, for reasons of economy, when doing spectral analyses using discrete Fourier transforms, we usually don't even compute the spectrum at frequencies above the folding frequency. Since it is always a mirror image of the spectrum below the folding frequency, we know what it looks like without even computing it.

Note: What happened to the peak structure:

In case you are wondering why the peaks in [Figure 9](#) have less structure than the peaks in [Figure 7](#), this is because the points at which I computed the spectral data in [Figure 9](#) were twice as far apart as the points at which I computed the spectral data in [Figure 7](#).

(The total frequency range in [Figure 9](#) is twice as wide as in [Figure 7](#), but I computed the same number of points in both cases.)

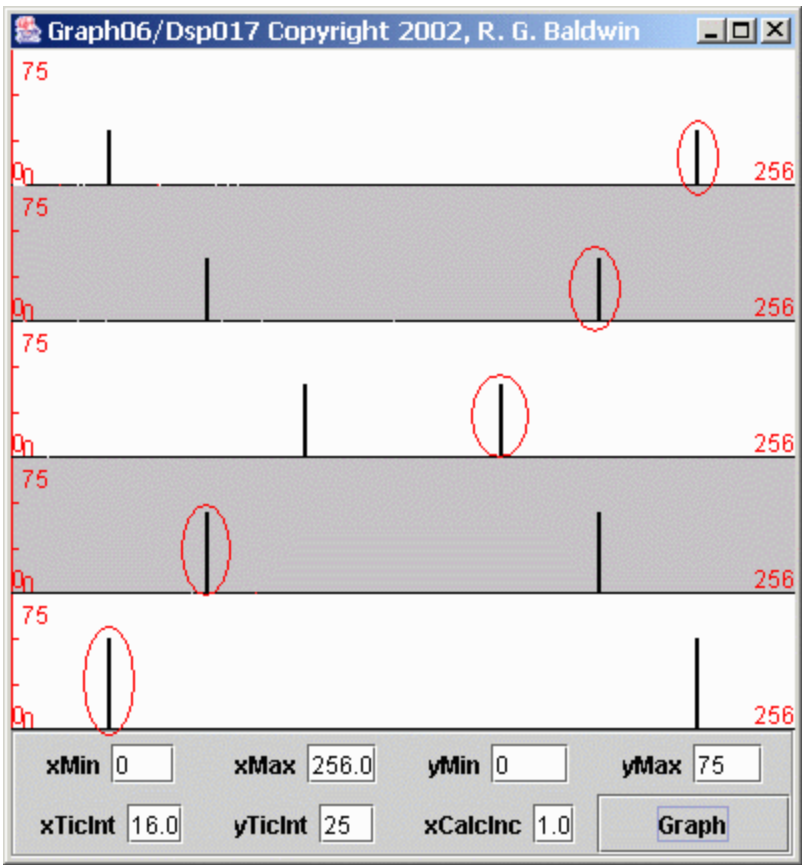
Although it's not obvious at this plotting scale, there are zero-valued points between the side lobes on the peaks in [Figure 7](#).

The points in the spectral display of [Figure 9](#) simply missed the side lobes and hit the zeros between the side lobes. I will have a lot more to say about this in a future module discussing spectral analysis.

Back to the case with the sampling problem

Now let's take another look at the case with the sampling problem. This is the case where the sampling frequency was reduced from four samples per second to two samples per second but the frequencies of the sinusoids was not changed. This view of the problem is shown in [Figure 10](#). It will probably be useful for you to compare [Figure 10](#) with [Figure 8](#).

Figure 10. Spectral analyses of five sinusoids with sampling problem



Folding frequency is at the center

As with the previous case, each of the plotted spectra in [Figure 10](#) shows the frequency range from zero frequency to the sampling frequency of two samples per second. The folding frequency of one cycle per second appears in the center of each plot.

Peaks to left correspond to [Figure 8](#)

The peaks to the left of center in [Figure 10](#) correspond to the peaks in [Figure 8](#). Because the right edge of [Figure 8](#) is the folding frequency, the peaks to the right of center in [Figure 10](#) don't appear in [Figure 8](#).

A mirror image

As is always the case, everything to the right of the folding frequency in [Figure 10](#) is a mirror image of everything to the left of the folding frequency.

I have identified the artifacts created by the sampling process with a red oval in [Figure 10](#).

Raw data frequency exceeds folding frequency

The problem, as you will recall, is that the frequency of the sinusoids corresponding to the two bottom plots in [Figure 10](#) is above the folding frequency. Thus the peaks to the right of center in the bottom two plots of [Figure 10](#) actually represent the frequencies of the corresponding sinusoids.

Unfortunately, these two peaks appear to the right of the folding frequency, which is the area of the spectra that we normally ignore.

Artifacts to the left of the folding frequency

Furthermore, these two peaks are reflected through the folded mirror image process into the area to the left of the folding frequency. For these two sinusoids, the peaks to the left of the folding frequency are artifacts, and I have identified them as such with ovals.

Normally can't identify artifacts

I am able to identify these two peaks as artifacts only because I know the true frequency makeup of the raw data. In most real-world situations with unknown data, there would be no way for me to identify these particular peaks to the left of the folding frequency as artifacts.

Illustrates the folding frequency

Hopefully this illustration will make the concept of the folding frequency easier for you to understand. The folding frequency is one-half the sampling frequency. The entire spectrum below the folding frequency folds around the folding frequency and the peaks in that spectrum appear in mirror-image format above the folding frequency.

The frequency information for all frequency components above the folding frequency is lost when the signal is sampled. In addition, the energy associated with those components will fold around and can corrupt the information for frequency components that are below the folding frequency.

The bottom line

The bottom line is that you must be very careful when sampling analog signals for later processing using DSP. In order to avoid erroneous results, you must sample sufficiently fast to ensure that your sampling rate is greater than twice the highest frequency components contained in the analog signal.

On the other hand, the greater your sampling rate, the more computer-intensive will be most of the DSP techniques that you apply to the data later. For economy reasons, therefore, you don't want your sampling frequency to be excessively high.

Using an analog low-pass pre-filter

A common approach to sampling is to feed the analog signal into an analog-to-digital (*AtoD*) converter. This is a device that measures the amplitude of the analog signal at a uniform sampling frequency. It is common practice to place a low-pass analog filter immediately ahead of the converter to suppress any analog frequency components that are greater than one-half the sampling frequency.

Digital re-sampling

Another common approach is to initially sample the analog signal at a sufficiently high rate to ensure that the sampling rate is greater than twice the highest frequency contained in the analog signal. Then, if you really don't need all of that high-frequency information, you can apply a low-pass digital filter to suppress the high-frequency energy. Then you can re-sample the data to a lower sampling frequency simply by discarding samples. The data with the lower sampling frequency can then be used for further DSP analysis.

Summary

I explained the meaning of sampling, and explained some of the problems that arise when sampling and processing analog signals.

The problems generally relate to the relationship between the sampling frequency and the high-frequency components contained in the analog signal.

I explained the concept of the Nyquist folding frequency, which is half the sampling frequency.

I illustrated the frequency folding phenomena by plotting sampled time series data as well as spectral data.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Dsp00104: Digital Signal Processing (DSP) in Java, Sampled Time Series
- File: Dsp00104.htm
- Published: 10/04/02

Baldwin explains the meaning of sampling, and identifies some of the problems that arise when sampling and processing analog signals. He explains the concept of the Nyquist folding frequency and illustrates the folding phenomena by plotting time series data as well as spectral data.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a

book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Dsp00108-Averaging Time Series

Baldwin begins with a discussion of averaging time series, and ends with a discussion of spectral resolution, covering several related topics in between.

Revised: Fri Oct 16 23:12:34 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
 - [Some of what you have previously learned](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Preview](#)
- [Discussion](#)
 - [Which screw to turn...](#)
 - [The mechanic and the screw](#)
 - [A very important module](#)
 - [Turning the screws in DSP](#)
 - [Computing the average value of a time series](#)
 - [Decomposition of time series](#)
 - [A sum of products of sinusoids](#)
 - [Many sinusoidal products](#)
 - [Examples of products of sinusoids](#)

- [More examples of the products of sinusoids](#)
- [What is the average value of a sinusoid?](#)
- [A short recap before continuing](#)
- [Spectral analysis](#)
 - [Several steps are involved](#)
 - [The Fourier transform](#)
 - [Let's see some data](#)
 - [A spectral line](#)
 - [Two spectral lines](#)
 - [Zero-valued points in the spectra](#)
 - [The frequency resolution of the Fourier transform](#)
- [Summary](#)
- [Miscellaneous](#)

Preface

This is one in a series of modules designed to teach you about Digital Signal Processing (DSP) using Java. The purpose of the miniseries is to present the concepts of DSP in a way that can be understood by persons having no prior DSP experience. However, some experience in Java programming would be useful. Whenever it is necessary for me to write a program to illustrate a point, I will write it in Java.

Some of what you have previously learned

In a previous module, I explained the meaning of sampling, and discussed some of the problems that occur as a result of high-frequency components in the analog signal.

Measure and record the signal amplitude

I told you that to sample an analog signal means to measure and record its amplitude at a series of points in time. The values that you record constitute

a *sampled time series* intended to represent the analog signal.

Avoiding frequency folding

I told you that to avoid problems, the sampling frequency must be at least twice as great as the highest frequency component contained in the analog signal, and as a practical matter, should probably be somewhat higher.

Sinusoids, frequency, and period

I introduced you to sinusoids, taught you about sine and cosine functions, and introduced the concepts of period and frequency for sinusoids.

Decomposition of time series

I told you that almost everything we will discuss in this series on DSP is based on the premise that every time series can be decomposed into a large number of sinusoids, each having its own amplitude and frequency.

The notion of DSP

I told you that DSP is based on the notion that signals in nature can be sampled and converted into a series of numbers. The numbers can be fed into some sort of digital device, which can process the numbers to achieve some desired objective.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

- [Figure 1](#). Products of sinusoids.

- [Figure 2](#). Products of sinusoids.
- [Figure 3](#). More products of sinusoids.
- [Figure 4](#). Five Sampled Sinusoids.
- [Figure 5](#). Computed average value of a time series.
- [Figure 6](#). Expanded average value of a time series.
- [Figure 7](#). Computed average value of a time series.
- [Figure 8](#). Computed average value of a time series.
- [Figure 9](#). Computed average value of a time series.
- [Figure 10](#). Spectra of five different sinusoids of different lengths.
- [Figure 11](#). Spectra of five different sinusoids of different lengths.
- [Figure 12](#). Spectra of five different time series of different lengths.
- [Figure 13](#). Spectra of five different time series of different lengths.
- [Figure 14](#). Average values of sinusoid products.
- [Figure 15](#). Illustration of frequency resolution.
- [Figure 16](#). Illustration of frequency resolution.

Preview

This is a broad-ranging module. It begins with a discussion of averaging time series, ends with a discussion of spectral resolution, and covers several related topics in between. Don't be alarmed, however, at the range of the module. The topics of time-series averaging and spectral resolution are very strongly related.

I will discuss why we frequently need to average sampled time series, and explain some of the issues involved in that process.

I will also show you the impact of those averaging issues on DSP, using spectrum analysis as an example.

Discussion

It never ceases to amaze me how something as mathematically complex as DSP can be distilled down to the simplest of computational processes.

Which screw to turn ...

The mechanic and the screw

DSP reminds me of the old story about the customer who complained about the bill at the auto repair shop being too high. According to the customer, all the mechanic did to fix the problem was turn one screw, and the bill was too high for the labor involved. The mechanic responded that he didn't charge for turning the screw. Instead, he charged for knowing which screw to turn, and knowing which way and how far to turn it.

A very important module

This module, in conjunction with the earlier module titled [Sampled Time Series](#) may be the most important module in the entire collection because it provides a practical pseudo-mathematical framework for almost everything that follows.

Almost everything that you will do using DSP involves:

1. Multiplying the sample values in one set of samples by the corresponding sample values in a second set of samples.
2. Computing the average of the set of multiplication products.
3. Interpreting the value of the average relative to the task at hand.

Once you understand the ramifications of the "multiply and average" process, the solution to many DSP problems simply involves figuring out how to index your way through the respective sample sets in order to apply the arithmetic appropriately. This is true for convolution, correlation, spectrum analysis, adaptive processing and many other forms of DSP as well.

Turning the screws in DSP

Knowing how to turn the screw is not the complicated part of DSP. Rather, the complicated part of DSP lies in knowing which screw to turn and which way to turn it. Once you know that, you will be surprised just how easy it is to actually turn the screw.

Computing the average value of a time series

As you will learn in this series of modules, a large majority of DSP operations consist simply of the following two steps:

1. Multiply one time series by another time series, to produce a third time series.
2. Compute the average value of the third time series.

In many cases, it is the average value of the third time series that provides the answer you are seeking.

The challenge is in knowing what the average value means, and how to interpret it.

Decomposition of time series

Almost everything that we will discuss in this series on DSP is based on the premise that every time series can be decomposed into a (*potentially large*) number of sinusoids, each having its own amplitude and frequency.

Suppose, for example, that we have two time series, each of which is composed of two sinusoidal components as follows:

$$\begin{aligned}f(x) &= \cos(ax) + \cos(bx) \\g(x) &= \cos(cx) + \cos(dx)\end{aligned}$$

The product of the two time series is given by:

$$\begin{aligned}h(x) &= f(x) * g(x) \\&= (\cos(ax) + \cos(bx)) * (\cos(cx) + \cos(dx))\end{aligned}$$

where the asterisk (*) means multiplication.

Multiplying this out produces the following:

$$\begin{aligned}h(x) = & \cos(ax) * \cos(cx) \\ & + \cos(ax) * \cos(dx) \\ & + \cos(bx) * \cos(cx) \\ & + \cos(bx) * \cos(dx)\end{aligned}$$

A sum of products of sinusoids

Thus, the time series produced by multiplying any two time series consists of the sum of a (*potentially large*) number of terms, each of which is the product of two sinusoids.

The product of two sinusoids

We probably need to learn a little about the product of two sinusoids. I will discuss this topic with a little more mathematical rigor in a future module. In this module, however, I will simply illustrate the topic using graphs.

Important: The product of two sinusoids is always a new time series, which is the sum of two new sinusoids.

The frequencies of the new sinusoids

The frequencies of the new sinusoids are different from the frequencies of the original sinusoids. Furthermore, the frequency of one of the new sinusoids may be zero.

Note: What is a sinusoid with zero frequency?

As a practical matter, a sinusoid with zero frequency is simply a constant value. It plots as a horizontal straight line versus time.

Think of it this way. As the frequency of the sinusoid approaches zero, the period, (*which is the reciprocal of frequency*), approaches infinity. Thus, the width of the first lobe of the sinusoid widens, causing every value in that lobe to be the same as the first value.

This will become a very important concept as we pursue DSP operations.

Sum and difference frequencies

More specifically, when you multiply two sinusoids, the frequency of one of the sinusoids in the new time series is the *sum of the frequencies* of the two sinusoids that were multiplied together. The frequency of the other sinusoid in the new time series is the *difference between the frequencies* of the two sinusoids that were multiplied together.

An important special case

For the special case where the two original sinusoids have the same frequency, the difference frequency is zero and one of the sinusoids in the new time series has a frequency of zero. It is this special case that makes digital filtering and digital spectrum analysis possible.

Many sinusoidal products

When we multiply two time series and compute the average of the resulting time series, we are in effect computing the average of the products of all the individual sinusoidal components contained in the two time series. That is, the new time series contains the products of (*potentially many*) individual sinusoids contained in the two original time series. In the end, it all comes down to computing the average value of products of sinusoids.

Product of sinusoids with same frequency

The product of any pair of sinusoids that have the same frequency will produce a time series containing the sum of two sinusoids. One of the sinusoids will have a frequency of zero (*hence it will have a constant value*). The other sinusoid will have a frequency that is double the frequency of the original sinusoids.

The ideal average value

Ideally, the average value of the new time series will be equal to the constant value of the sinusoid with zero frequency. This is because, ideally, the average value of the other sinusoid will be zero.

Product of sinusoids with different frequencies

The product of any pair of sinusoids that do not have the same frequency will produce a new time series containing the sum of two sinusoids. One of the new sinusoids will have a frequency that is the sum of the frequencies of the two original sinusoids. The other sinusoid will have a frequency that is the difference between the frequencies of the two original sinusoids.

Ideal average value is zero

Ideally, the average value of the new time series in this case will be equal to zero, because ideally the average value of each of the sinusoids that make up the time series will be zero.

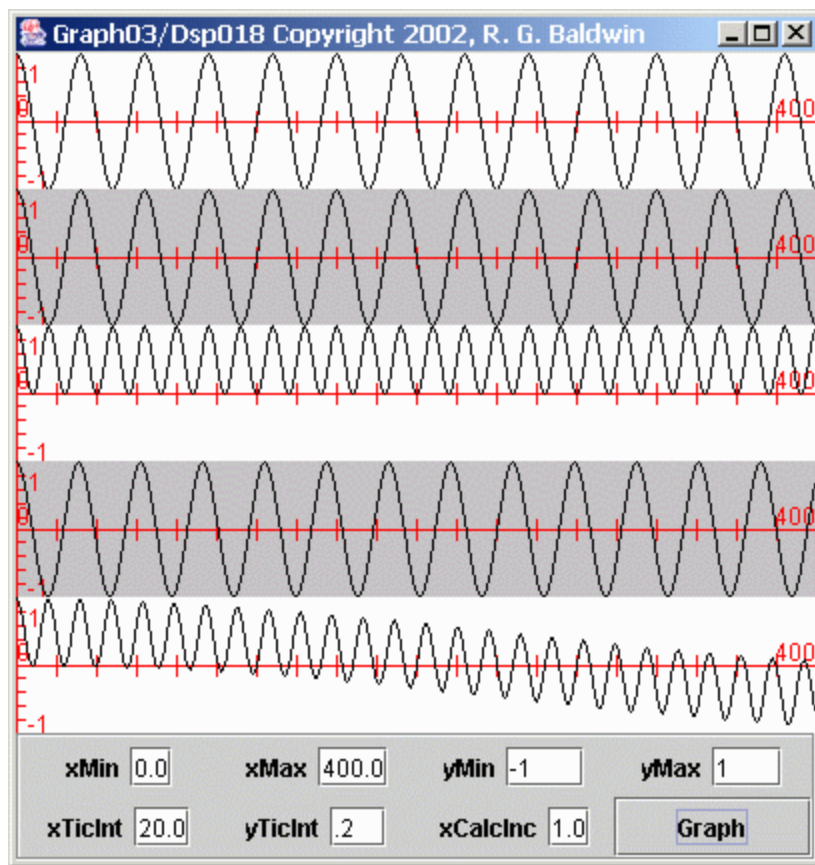
Oops!

As we will see later, we don't always achieve the ideal.

Examples of products of sinusoids

Let's examine some time series produced by multiplying sinusoids. [Figure 1](#), [Figure 2](#), and [Figure 3](#) show the results of multiplying sinusoids having the same and different frequencies. Consider first the plots in [Figure 1](#).

Figure 1. Products of sinusoids.



Multiplying sinusoids with same frequency

The top plot in [Figure 1](#) shows a sinusoid whose frequency and sampling rate are such that it has 32 samples per cycle. The second plot from the top in [Figure 1](#) is identical to the top plot. *(To simplify the explanation, these two sinusoids are also cosine functions.)*

The third plot down from the top in [Figure 1](#) shows the product of these two sinusoids, which have the same frequency. If you examine the third plot, you will notice several important characteristics.

A double-frequency sinusoid

By matching the peaks, you can determine that the frequency of the sinusoid in the third plot is double the frequency of each of the top two plots. *(This is the sum of the frequencies of the two sinusoids that were multiplied together.)*

Half the amplitude with a positive bias

Next, you will notice that the amplitude of the sinusoid in the third plot is half that of each of the first two plots. In addition, the entire sinusoid in the third plot is in the positive value range.

The sum of two sinusoids

The third plot is actually the sum of two sinusoids. One of the sinusoids has a frequency of zero, giving a constant value of 0.5. This constant value of 0.5 is added to all the values in the other sinusoid, causing it to be plotted in the positive value region.

Later on, we will compute the average value of the time series in the third plot. Ideally, that average value will be the constant value produced by the zero-frequency sinusoid.

Product of sinusoids with different frequencies

Now consider the bottom two plots in [Figure 1](#). The fourth plot down from the top is a cosine function whose frequency is almost, but not quite the same as the frequency of the sinusoid in the top plot. The sinusoid in the top plot has 32 samples per cycle while the sinusoid in the fourth plot has 31 samples per cycle.

The time series in the bottom plot is the product of the time series in the first and fourth plots.

The sum of two sinusoids

Once again, this time series is the sum of two sinusoids. The frequency of one is the difference between the two original frequencies. The frequency of the other is the sum of the two original frequencies.

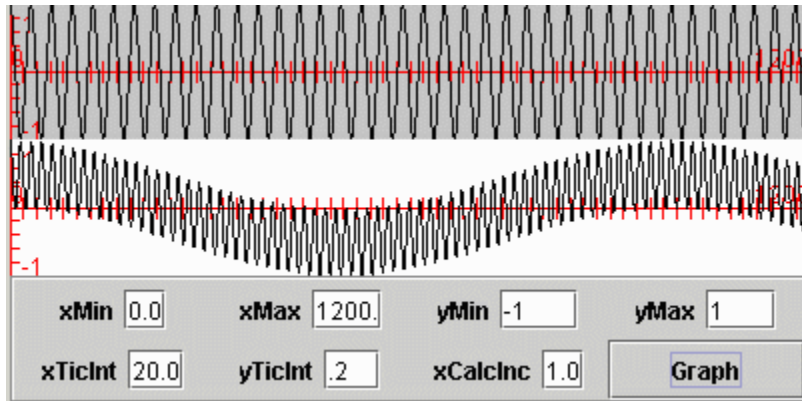
However, in this case, the difference frequency is **not zero**. Rather, it is a very low frequency. What you see in the bottom plot of [Figure 1](#) is a sinusoid whose frequency is the sum of the two original frequencies added to a sinusoid whose frequency is the difference between the two original frequencies. Because the two original frequencies were almost equal, the frequency of the second sinusoid is very low.

As you can see, the low-frequency component in the bottom plot in [Figure 1](#) appears to be the beginning of a cosine function whose period is much greater than the width of the plot (*400 points*).

Another view of the same data

[Figure 2](#) shows another view of the bottom two plots from [Figure 1](#).

Figure 2. Products of sinusoids.



The difference between [Figure 2](#) and [Figure 1](#) is that while [Figure 1](#) shows only 400 points along the x-axis, [Figure 2](#) shows 1200 points along the x-axis. Thus, the horizontal scale in [Figure 2](#) is significantly compressed relative to the horizontal scale in [Figure 1](#).

More than one cycle

[Figure 2](#) lets you see a little more than one full cycle of the low-frequency component of the time series produced by multiplying the two sinusoids.

([Figure 2](#) does not provide a very good representation of the high-frequency component. This is because I plotted 1200 points in a part of the screen that is only 400 pixels wide. On my computer, I can expand this to the full screen width. However, I can't publish it at that width, so I published the 400-pixel version.)

Averaging can be problematic in this case

Later on, we will compute the average value of the time series represented by the bottom plot in Figures 1 and 2. Ideally, that average value will be zero. However, you have probably already figured out that a great many data points must be included in the computation of the average to get anything near zero. An eyeball estimate indicates that about 900 data points are required just to include a single cycle of the low-frequency component.

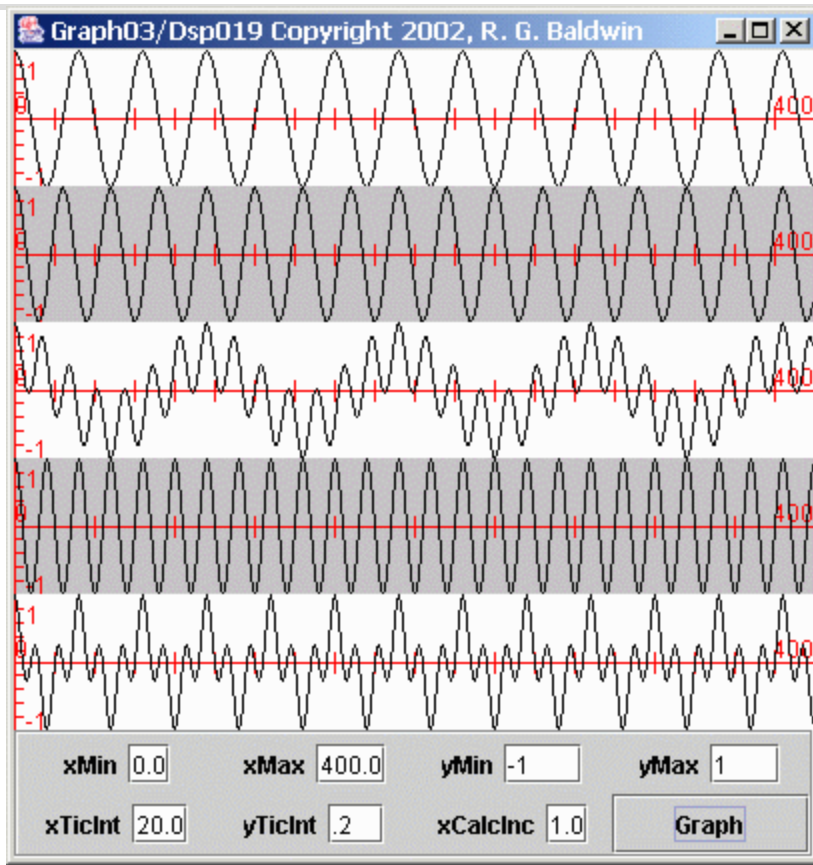
More examples of the products of sinusoids

[Figure 3](#) shows two additional time series created by multiplying sinusoids.

Figure 3. More products of sinusoids.



Figure 3. More products of sinusoids.



The arrangement in [Figure 3](#) is the same as in [Figure 1](#). The top plot in [Figure 3](#) is the same sinusoid shown in the top plot of [Figure 1](#). This is a sinusoid with 32 samples per cycle.

Immediately below the top sinusoid in [Figure 3](#) is another sinusoid. This sinusoid has 24 samples per cycle. As you can see, the frequency of this sinusoid is a little higher than the frequency of the sinusoid in the top plot.

The time series in the third plot down from the top is the product of the time series in the top two plots. Again, this time series is composed of two new sinusoids whose frequencies are the sum of and difference between the two original frequencies.

A greater frequency difference

Because the frequency difference between the first two plots in [Figure 3](#) is considerably greater than was the case for the bottom plot of [Figure 1](#), the frequency of the low frequency component of the third plot in [Figure 3](#) is considerably greater than was the case in [Figure 1](#).

Later on, we will compute the average value of the third plot in [Figure 3](#). Ideally, the average value will be zero.

An even greater frequency difference

The fourth plot in [Figure 3](#) shows a sinusoid having 16 samples per cycle. The frequency of this sinusoid is double the frequency of the sinusoid in the top plot.

The bottom plot in [Figure 3](#) shows the product of the first and fourth plots. As usual, this time series consists of the sum of two sinusoids whose frequencies are the sum and the difference of the original frequencies.

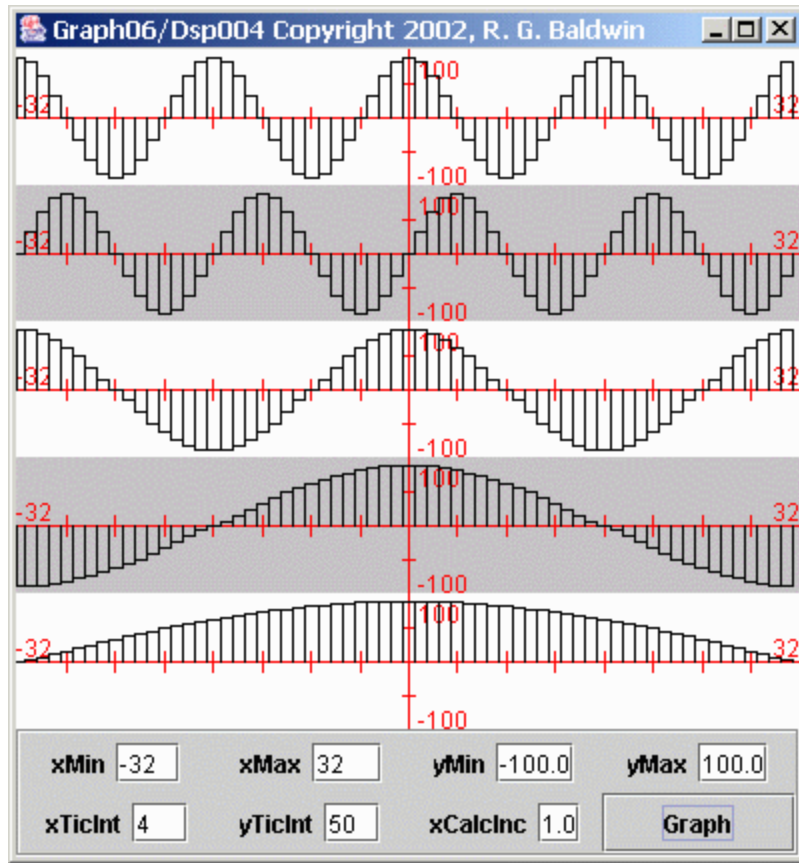
We will also compute the average value of the bottom plot later on. Ideally, the average value will be zero.

What is the average value of a sinusoid ?

Consider [Figure 4](#), which shows five different sampled sinusoids with different frequencies.

Figure 4. Five Sampled Sinusoids.

Figure 4. Five Sampled Sinusoids.



We may be tempted to say that the average value of a sinusoid is zero. After all, the positive lobes of the sinusoid are shaped exactly like the negative lobes. *Therefore, every positive value is offset by a corresponding negative value.*

Is that a true statement?

Every positive value is offset by a corresponding negative value only if you compute the average over an even number of cycles of the sinusoid. For example, it is pretty obvious that if you compute the average on the 64 data values shown for the bottom plot in [Figure 4](#), the result will not be zero. Rather, it will be a positive non-zero value.

Important: While the theoretical average value of a sinusoid is zero, the actual computed average value of a sinusoid will be zero only if you include an even number of cycles in the data used to compute the average.

Sample average values

Next we will take a look at the computed average values of the time series from Figures 1 and 3 that were produced by multiplying sinusoids.

The black curve in [Figure 5](#) shows an expanded view of the sinusoidal curve from the top half of the third plot in [Figure 1](#) (recall that the bottom half of that plot was empty, so I didn't include it in [Figure 5](#)). This curve was the result of multiplying two sinusoids with the same frequency.

Figure 5. Computed average value of a time series.

[missing_resource: file:///M:/Baldwin/AA-School/Connexions/Digital%20Signal%20Processing%20-%20DSP/2-Time%20Series/3-Dsp00108-Averaging%20Time%20Series/dsp00108fige.gif]

The computed average value

The red curve in [Figure 5](#) shows the computed average value as a function of the number of points included in the average. In other words, a particular point on the red curve in [Figure 5](#) represents the average value of all the

points on the black curve to the left of and including that point on the black curve.

The blue horizontal line in [Figure 5](#) shows the ideal average value for this situation.

Result converges on the ideal

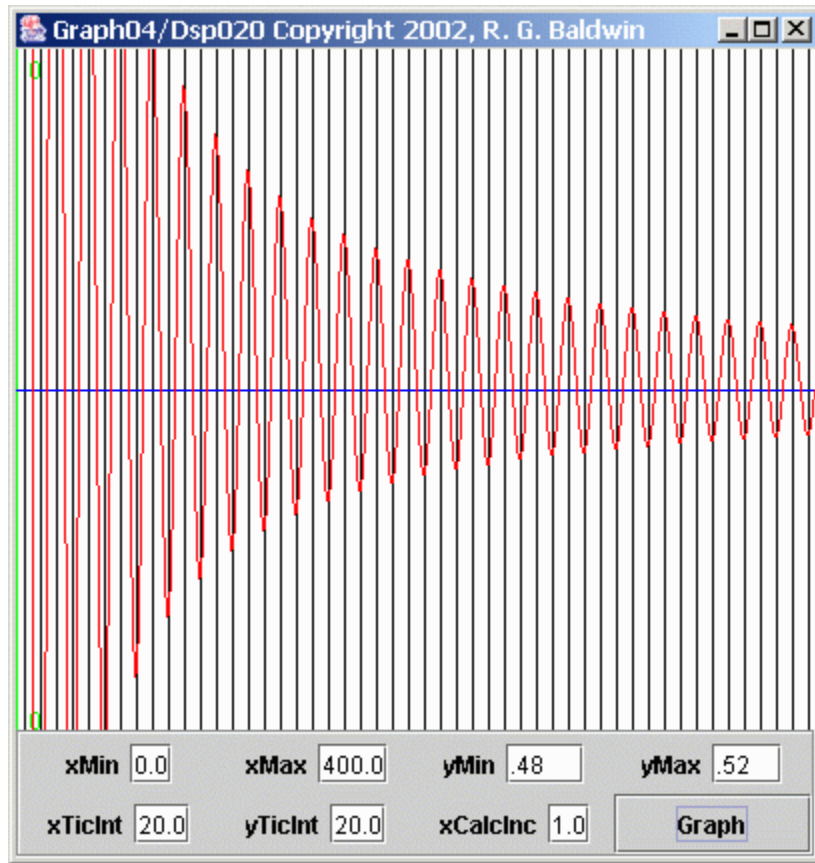
As more and more points are included in the average, the values of the positive and negative peaks on the red curve approach the ideal blue line asymptotically (*except for a slight positive bias, which is the result of the sampling process*).

An expanded view

[Figure 6](#) shows a greatly expanded view of the red average values in [Figure 5](#).

Figure 6. Expanded average value of a time series.

Figure 6. Expanded average value of a time series.



The ideal value for this average is 0.5, and that is the value represented by the blue line. The plot in [Figure 6](#) shows the same horizontal scale as [Figure 5](#). However, the entire vertical plotting area in [Figure 6](#) represents the values from 0.48 to 0.52.

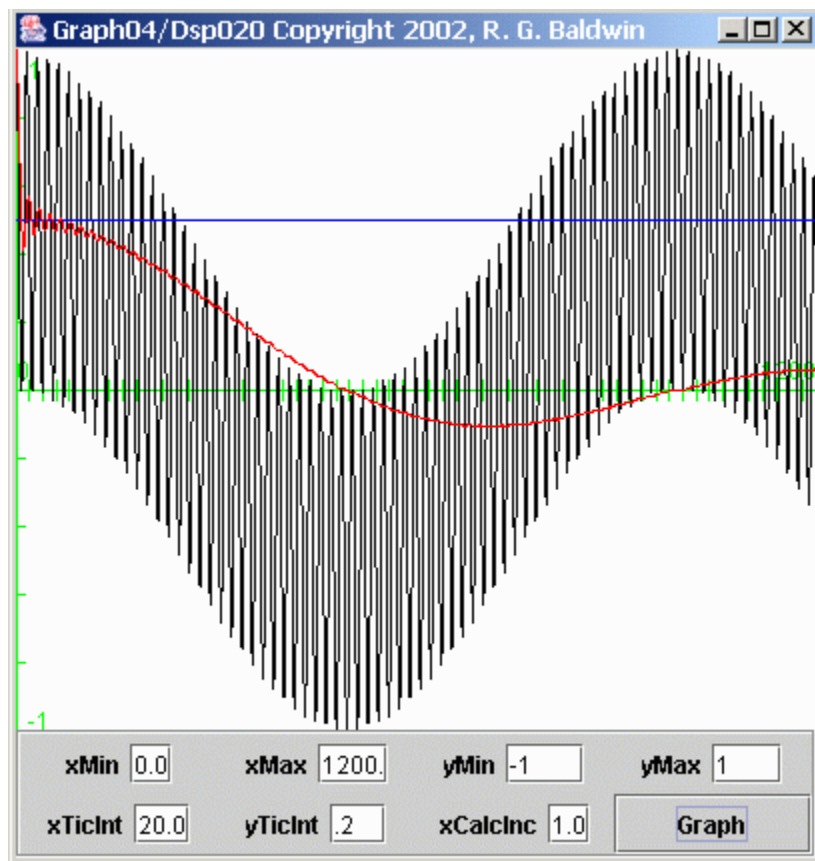
Ideal value is never reached

As you can see, the ideal value is never reached in [Figure 6](#) except at isolated points where the red curve crosses the horizontal line. Even if I extended the horizontal axis to 1200 or more points, that would continue to be the case.

A more serious case

[Figure 7](#) computes and displays the average value of the bottom plot in [Figure 2](#) (recall that this plot shows 1200 points on the horizontal axis, whereas [Figure 5](#) shows only 400 points on the horizontal axis). Recall also that this time series was produced by multiplying two sinusoids having nearly the same but not exactly the same frequency.

Figure 7. Computed average value of a time series.



Red curve is the average

As before, the black curve in [Figure 7](#) shows the time series, and the red curve shows the computed average value as a function of the number of points included in the average.

(In this case, I didn't even bother to show the short axis containing only 400 points. The horizontal axis in [Figure 7](#) contains 1200 points, the same as in [Figure 2](#).)

The ideal average value is zero

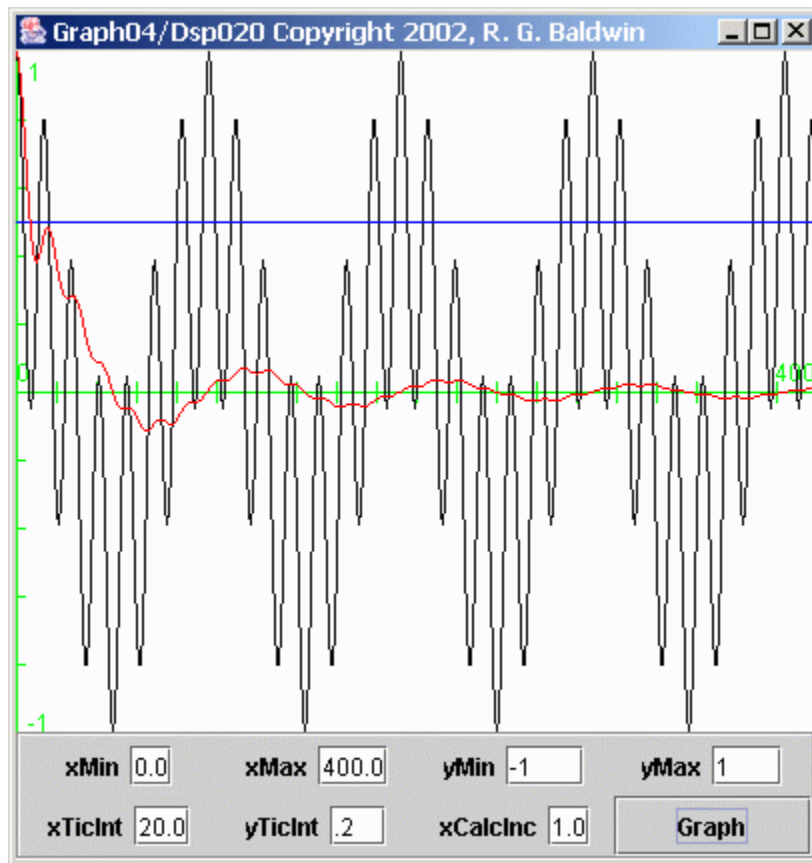
In this case, the ideal average value is zero, as indicated by the green horizontal axis. As you can see, even for a 1200-point averaging window, the average value deviates significantly from the ideal. We will see the detrimental impact of this problem later when I perform spectral analysis in an attempt to separate two closely-spaced peaks in the frequency spectrum.

Some additional examples of average values

[Figure 8](#) computes and displays the average value of the third plot down from the top in [Figure 3](#). This plot was produced by multiplying the two sinusoids in the top two plots in [Figure 3](#).

Figure 8. Computed average value of a time series.

Figure 8. Computed average value of a time series.

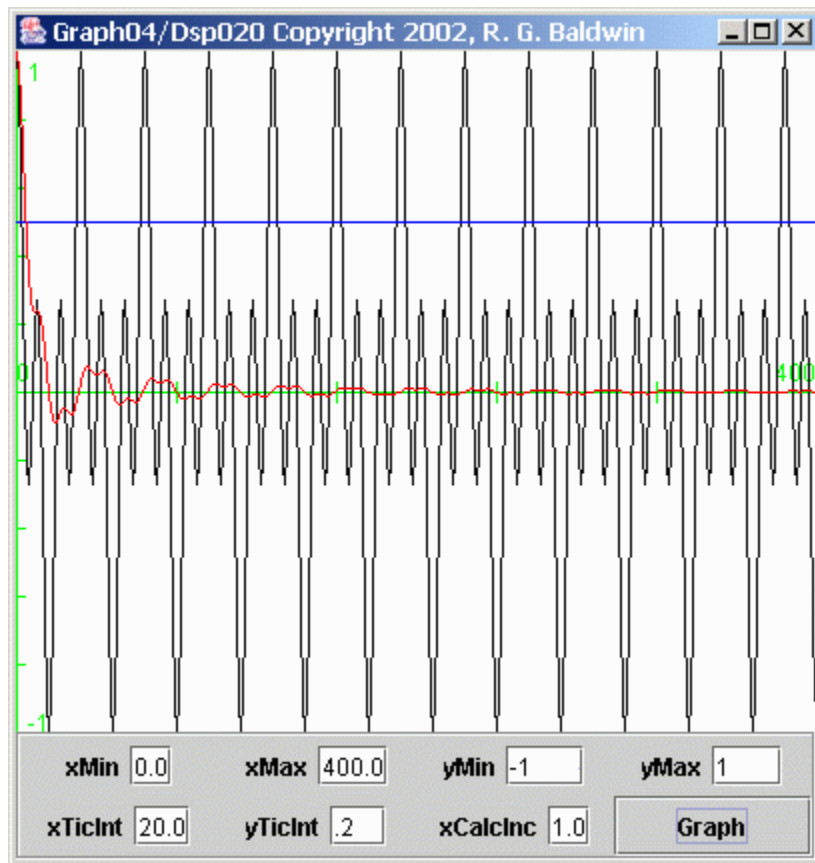


As before, the black curve in [Figure 8](#) represents the time series, and the red curve represents the computed average value of the time series as a function of the number of points included in the average.

For this case also, the ideal average value is zero, as represented by the green horizontal axis. The positive and negative peaks in the red average value can be seen to approach the ideal value asymptotically within the 400 horizontal points plotted in [Figure 8](#).

[Figure 9](#) computes and displays the average value of the bottom plot in [Figure 3](#). This time series was produced by multiplying the top plot in [Figure 3](#) by the fourth plot in [Figure 3](#).

Figure 9. Computed average value of a time series.



Once again, the black curve in [Figure 9](#) represents the time series, and the red curve represents the computed average value of the time series as a function of the number of points included in the average. In this case, the average converges on zero rather nicely within the 400 points included on the horizontal axis.

A short recap before continuing

Hopefully, by this point, you understand how multiplying two time series produces a new time series composed of the sum of all the products of the individual sinusoids in the two original time series.

When each pair of sinusoids is multiplied together, they produce a new time series consisting of two other sinusoids whose frequencies are the sum and difference of the original pair of frequencies.

The error in the computed average

When an average is computed for a fixed number of points on the new time series, the error in the average tends to be greater for cases where the original frequency values were close together. This is because the period of one of the new sinusoids becomes longer as the original frequencies become closer. In general, the longer the period of the sinusoid, the more points are required to get a good estimate of its average value.

Does this matter?

There are many operations in DSP where this matters a lot. As mentioned earlier, the computational requirements for DSP frequently boil down to nothing more than multiplying a pair of time series and computing the average of the product. You will see many examples of this as you continue studying the modules in this series of tutorials on DSP.

Spectral analysis

I am going to illustrate my point by showing you one such example in this module. This example will use a Fourier transform in an attempt to perform spectral analysis and to separate two closely-spaced frequency components in a time series. As you will see, errors in the computed average can interfere with this process in a significant way.

(This example will illustrate and explain the results using graphs. Future modules will provide more technical details on the DSP operations involved.)

Several steps are involved

I will provide this illustration in several steps.

Spectral data for same frequency but different lengths

First, I will show you spectral data for several time series, each consisting of a single sinusoid. The time series will have different lengths but the individual sinusoids will have the same frequency. This will serve as baseline data for the experiments that follow.

Sum of two sinusoids

Then I will show you spectral data for several time series, each composed of the sum of two sinusoids. These time series will have different lengths. The sinusoids in each time series will have the same frequencies. I will show you two cases that fall under this description. The frequency difference for the two sinusoids in each time series will be small in one case, and greater in another case.

Sinusoids with different frequency differences

Finally, I will show you spectral data for several time series, each composed of the sum of two sinusoids. These time series will be different lengths, and the sinusoids in each time series will have different frequencies. In particular, the frequency difference between the two sinusoids in each time series will be equal to the theoretical *frequency resolution* for a time series of that particular length.

The Fourier transform

In order to perform the spectral analysis, I will perform a Fourier transform on the time series to transform that data into the frequency domain. Then I will plot the data in the frequency domain.

(This module will not provide technical details on the Fourier transform. That information will be forthcoming in a future module.)

Keeping it simple

To keep this explanation as simple as possible, I will stipulate that all of the sinusoids contained in the time series are cosine functions. There are no sine functions in the time series.

(If the time series did contain sine functions, the process would still work, but the explanation would be more complicated.)

A brief description of the Fourier transform

Before I get into the results, I will provide a very brief description of how I performed the Fourier transform for these experiments.

The following steps were performed at each frequency in a set of 400 uniformly spaced frequencies across the frequency range from zero to the folding frequency.

The steps were:

- If the time series was shorter than 400 points, extend it to 400 points by appending zero-valued points at the end.

- Select the next frequency of interest.
- Generate a cosine function, 400 samples in length, at that frequency.
- Multiply the cosine function by the time series.
- Compute the average value of the time series produced by multiplying the cosine function by the time series.
- Save the average value. Call it the *real* value for later reference.
- Generate a sine function, 400 samples in length, at the same frequency.
- Multiply the sine function by the time series.
- Compute the average value of the time series produced by multiplying the sine function by the time series.
- Save the average value. Call it the *imaginary* value for later reference.
- Compute the square root of the sum of the squares of the real and imaginary values. This is the value of interest. Plot it.

Why does this work?

No matter how many sinusoidal components are contained in the time series, only one (*if any*) of those sinusoidal components will match the selected frequency.

Multiply by the cosine and average the product

When that matching component is multiplied by the cosine function having the selected frequency, the new time series created by the multiplication will consist of a constant value plus a sinusoid whose frequency is twice the selected frequency.

The computed average value of this time series will converge on the value of the constant with the quality of the estimate depending on the number of points included in the average.

Multiply by the sine and average the product

Since the sinusoids in the time series are stipulated to be cosine functions, when the sinusoid with the matching frequency is multiplied by the sine function, the new time series will consist of a constant value of zero plus a sinusoid whose frequency is twice the frequency of the sine function.

The computed average of this time series will converge on zero with the quality of the estimate depending on the number of points in the average.

(As mentioned earlier, this process would work even if the time series contained sinusoids other than cosine functions. However, the explanation would be more complicated.)

What about the other sinusoidal components?

Every other sinusoidal component in the time series (*whose frequency doesn't match the selected frequency*), will produce a new time series containing two sinusoids when multiplied by the sine function or the cosine function.

The frequency of one of the sinusoids in the new time series will be the sum of the frequencies of the sinusoidal component and the sine or cosine function. The frequency of the other sinusoid will be the difference in the frequencies between the sinusoidal component and the sine or cosine function.

As you saw earlier, when this difference is very small, the frequency of the new sinusoid will be very near to zero.

The average value for non-matching components

Ideally, the average value of the product should be zero when the frequency of the original sinusoidal component is different from the sine or cosine function by which it is multiplied. The computed average of this time series

will converge on zero with the quality of the estimate depending on the number of points in the average.

Measurement error

However, (*and this is very important*), when the frequency of the original sinusoid is very close to the frequency of the sine or cosine function, the convergence on zero will be poor even for a large number of points in the average.

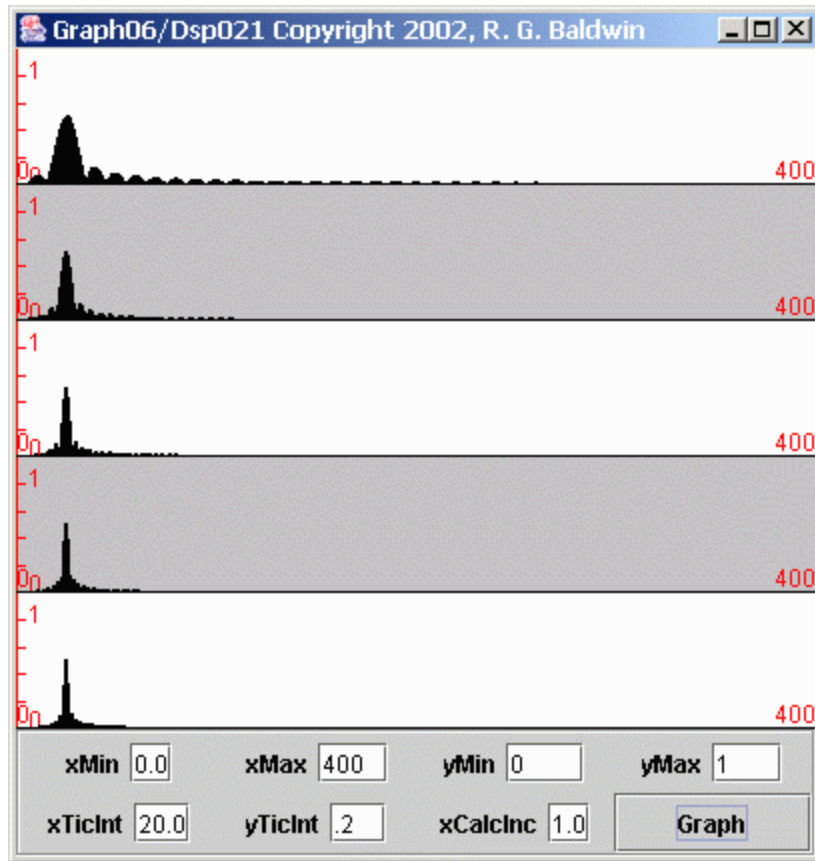
Thus, the computation at those frequencies very near to the frequency of an actual sinusoidal component in the raw data will produce a non-zero average value even when there is no sinusoidal component in the raw data at those frequencies. This is a form of measurement error.

Let's see some data

With that as a preface, let's look at some graphs ([Figure 10](#) and [Figure 11](#)) resulting from spectral analyses. (*These two figures show two different views of the same data.*)

Figure 10. Spectra of five different sinusoids of different lengths.

Figure 10. Spectra of five different sinusoids of different lengths.



Five sinusoids, same frequency, different lengths

[Figure 10](#) shows the individual spectra computed for five different sinusoids, each having the same frequency, but different lengths. The combination of sampling rate and frequency was such that each sinusoid had 32 samples per cycle.

Starting at the top in [Figure 10](#), the lengths of the five sinusoids were 80, 160, 240, 320, and 400 samples. (*The lengths of the five sinusoids were multiples of 80 samples.*)

Extend to 400 samples for computation

As mentioned earlier, for the cases where the actual length of the sinusoid was less than 400 samples, the length was extended to 400 samples by appending an appropriate number of samples having zero values.

(This made it easy to compute and plot the spectrum for every sinusoid over the same frequency range with the same number of points in each plot.)

The spectrum was computed and plotted for each sinusoid at 400 individual frequency points between zero and the folding frequency.

The actual averaging window

Even though the Fourier transform program averaged across 400 samples in all cases, the *effective* averaging length was equal to the length of the sinusoid. All product points outside that length had a value of zero and contributed nothing to the average one way or the other.

(I also applied an additional scale factor to the spectral results to compensate for the fact that fewer total samples were included in the average for the short samples. This caused the amplitude of the peak in the spectrum to be nominally the same in all five cases.)

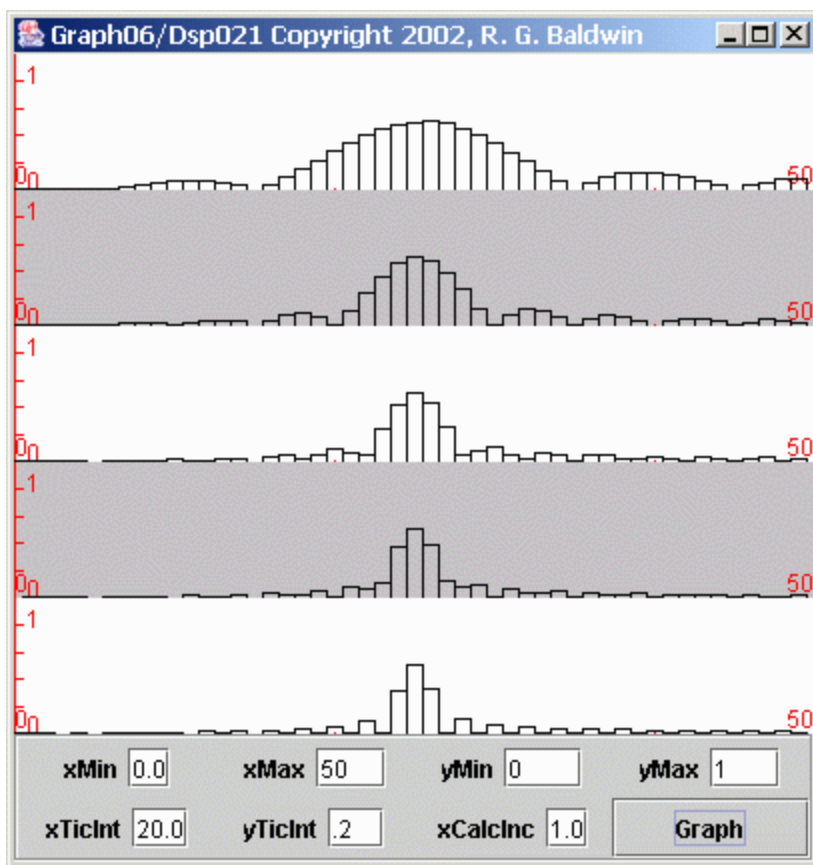
A horizontally-expanded plot

As you can see in [Figure 10](#), there isn't much in the spectra to the right of about 50 spectral points. That is as it should be since the single sinusoid in

each time series was at the low end of the spectrum.

[Figure 11](#) shows the same data as [Figure 10](#) with only the first 50 frequency points plotted on the horizontal axis. The remaining 350 frequency points were simply ignored. This provides a much better view of the structure of the peaks in the different spectra.

Figure 11. Spectra of five different sinusoids of different lengths.



I will begin the discussion with the bottom plot in [Figure 11](#), which is the computed spectrum for the single sinusoid having a length of 400 samples.

A spectral line

Ideally, since the time series was a single sinusoid, the spectrum should consist of a single non-zero value at the frequency of the sinusoid, (*often referred to as a spectral line*) and every other value in the spectrum should be zero.

However, because the computation of the spectrum involves the computation of average values resulting from the products of sinusoids, the ideal is not always achieved. In order to achieve the ideal, it would be necessary to multiply and average over an infinite number of points. Anything short of that will result in some measurement error, as exhibited by the bottom plot in [Figure 11](#).

(The bottom plot in [Figure 11](#) has a large peak in the center with every second point to the left and right of center having a zero value. I will explain this structure in more detail later.)

Spectra of shorter sinusoids

Moving from the bottom to the top in [Figure 11](#), each individual plot shows the result of shorter and shorter averaging windows. As a result, the measurement error increases and the peak broadens for each successive plot going from the bottom to the top in [Figure 11](#). The plot at the top, with an averaging window of only 80 samples, exhibits the most measurement error and the broadest peak.

(It should be noted, however, that even the spectra for the shorter averaging windows have some zero-valued points. Once you understand the reason for the zero-valued points, you can correlate the positions of those points to the length of the averaging windows in [Figure 11](#).)

Two spectral lines

Now I'm going to show you the detrimental impact of such spectral measurement errors. In particular, the failure of the average to converge on zero for short averaging windows limits the spectral resolution of the Fourier transform.

Five time series with two sinusoids each

I will create five new time series, each consisting of the sum of two sinusoids with fairly closely-spaced frequencies. One sinusoid has 32 samples per cycle as in Figures 10 and 11. The other sinusoid has 26 samples per cycle.

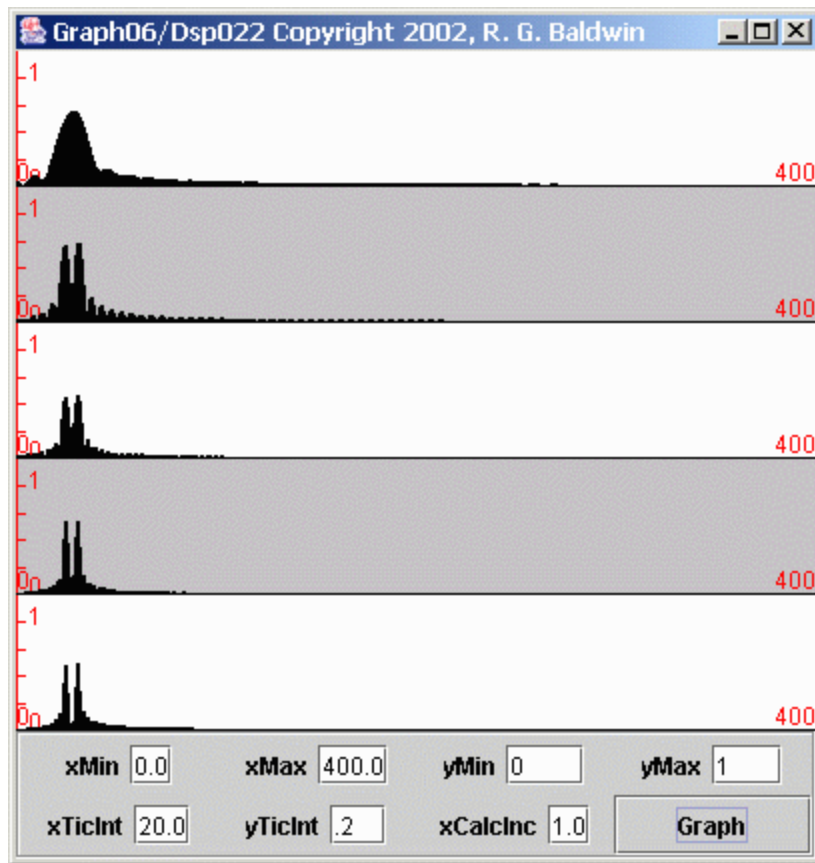
As before, the lengths of the individual time series will be 80, 160, 240, 320, and 400 samples respectively.

Spectral analysis using Fourier transform

I will perform a Fourier transform on each of the time series in an attempt to show that the spectrum of each time series consists of two peaks, (*two spectral lines*) , with one peak corresponding to each of the sinusoids added together to create the time series. The five spectra are shown in [Figure 12](#).

Figure 12. Spectra of five different time series of different lengths.

Figure 12. Spectra of five different time series of different lengths.



Discuss the longest time series first

Once again, let's begin with the plot at the bottom of [Figure 12](#). As you can see, this spectrum shows two very distinct spectral peaks. Thus, for this amount of frequency separation and a length of 400 samples, the Fourier transform did a good job of separating the two peaks.

Resolution for shorter averaging windows

Moving upward in [Figure 12](#), we see that the Fourier transform on the time series with a length of 320 samples (*the fourth plot from the top*) also did a good job of separating the two peaks.

However, with respect to separation the process began to deteriorate for lengths of 240 samples and 160 samples.

No peak separation for 80-sample average

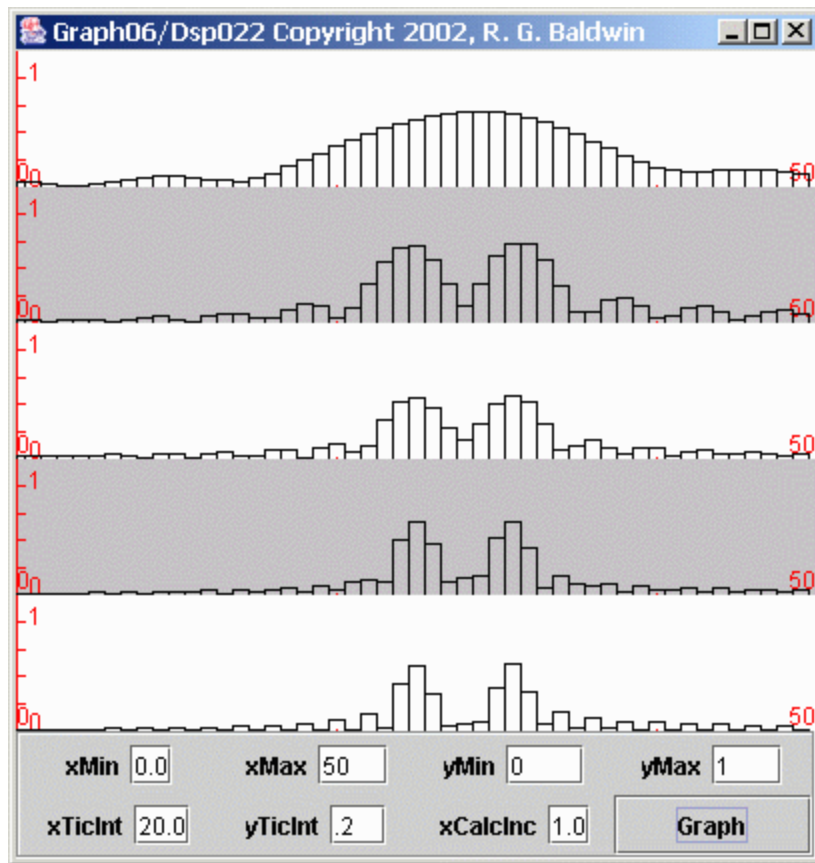
For a length of 80 samples, the two peaks merged completely.

A horizontally-expanded view of the spectra

[Figure 13](#) shows a horizontally-expanded view of the same spectral data to give you a better idea of the structure of the peaks. The plots in [Figure 13](#) show only the first fifty frequency values.

Figure 13. Spectra of five different time series of different lengths

Figure 13. Spectra of five different time series of different lengths



You may find it interesting to make a side-by-side comparison of Figures 13 and 11 in separate browser windows.

Zero-valued points in the spectra

Before leaving this topic, there are a few more things that I want to show you. If you go back and look at the bottom plot in [Figure 11](#), you will note an interesting characteristic of that plot. In particular, starting at the peak and moving outward in both directions, every second plotted value is zero. I'm going to explain the reason for and the significance of this characteristic.

(As I mentioned earlier, there are also zero-valued points in the spectra of the time series with the shorter averaging windows. Once you understand the reason for the zero-valued points, you can correlate the positions of those points to the length of the averaging window.)

400 spectral values were computed

To begin with, the Fourier transform program that was used to compute this spectrum computed 400 values at equally spaced points between zero and the folding frequency (*only the first 50 values are shown in [Figure 11](#)*). Thus, each of the side-by-side rectangles in [Figure 11](#) represents the spectral value computed at one of the 400 frequency points.

Sampling frequency was one sample per second

The sinusoid that was used as the target for this spectral analysis had 32 samples per cycle. Since this sinusoid was generated mathematically instead of being the result of sampling an analog signal, we can consider the sampling frequency to be anything that we want.

For simplicity, let's assume that the sampling frequency was one sample per second. This causes the sinusoid to have a period of 32 seconds and a frequency of 0.03125 cycles per second.

At a sampling rate of one sample per second, the folding frequency occurs at 0.5 cycles per second.

The computational frequency interval

Dividing the folding frequency by 400 we conclude that the Fourier transform program computed a spectral value every 0.00125 cycles per

second. Given that every second spectral value is zero, the zero values occur every 0.00250 cycles per second.

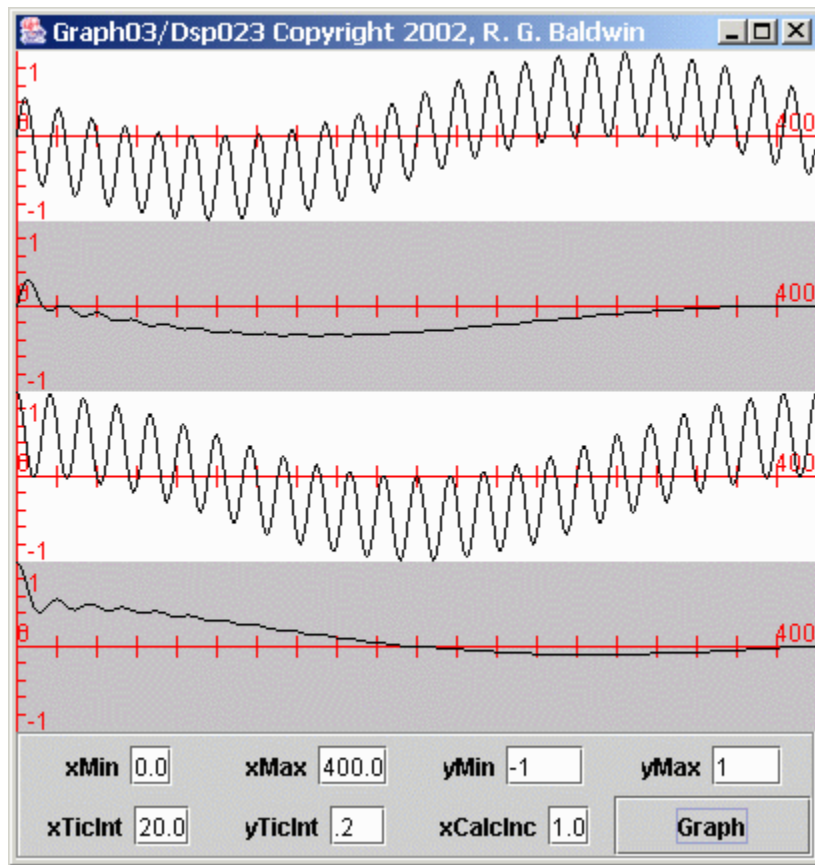
Let's compute the average of some products

The top plot in [Figure 14](#) shows the result of multiplying a cosine function having a frequency of 0.03125 cycles per second (*the frequency of the sinusoid in the previous spectral analysis experiment*) by a sine function having a frequency of 0.02875 cycles per second.

(This replicates one of the steps in the computation of the imaginary value in the Fourier transform).

Figure 14. Average values of sinusoid products.

Figure 14. Average values of sinusoid products.



The difference between the frequencies of the cosine function and the sine function is 0.00250 cycles per second.

(Note that this frequency difference is the reciprocal of the actual number of samples in the earlier time series, which contained 400 samples. This is also the frequency interval on which the Fourier transform produced zero-valued points for the bottom plot in [Figure 11](#).)

The average of the product time series

The second plot in [Figure 14](#) shows the average value of the time series in the first plot versus the number of samples included in the averaging window.

(This replicates another step in the computation of the imaginary value in the Fourier transform).

It is very important to note that this average plot goes to zero when 400 samples are included in the average.

Product of two cosine functions

Similarly, the third plot in [Figure 14](#) shows the product of the same cosine function as above and another cosine function having the same frequency as the sine function described above.

(This replicates a step in the computation of the real value in the Fourier transform).

The average of the product time series

The fourth plot in [Figure 14](#) shows the average value of the time series in the third plot.

(This replicates another step in the computation of the real value in the Fourier transform).

This average plot goes to zero at an averaging window of about 200 samples, and again at an averaging window of 400 samples.

Where do the zero values match?

The first point at which both average plots go to zero at the same point on the horizontal axis is at an averaging window of 400 samples.

(Both the real and imaginary values must go to zero in order for the spectral value produced by the Fourier transform to go to zero.)

Zero values in the spectrum for a sinusoid

Thus, the values produced by performing a Fourier transform on a single sinusoid go through zero at regular frequency intervals out from the peak in both directions. The frequency intervals between the zero values are multiples of the reciprocal of the actual length of the sinusoid on which the transform is performed.

(Note however, that you may not see the zero-valued points in the spectrum if you don't compute the spectral values at exactly those frequency values. This is the case for some of the plots in [Figure 11](#).)

The frequency resolution of the Fourier transform

Some analysts regard a frequency interval equal to the reciprocal of the length of the time series as being the useful resolution of the spectrum analysis process.

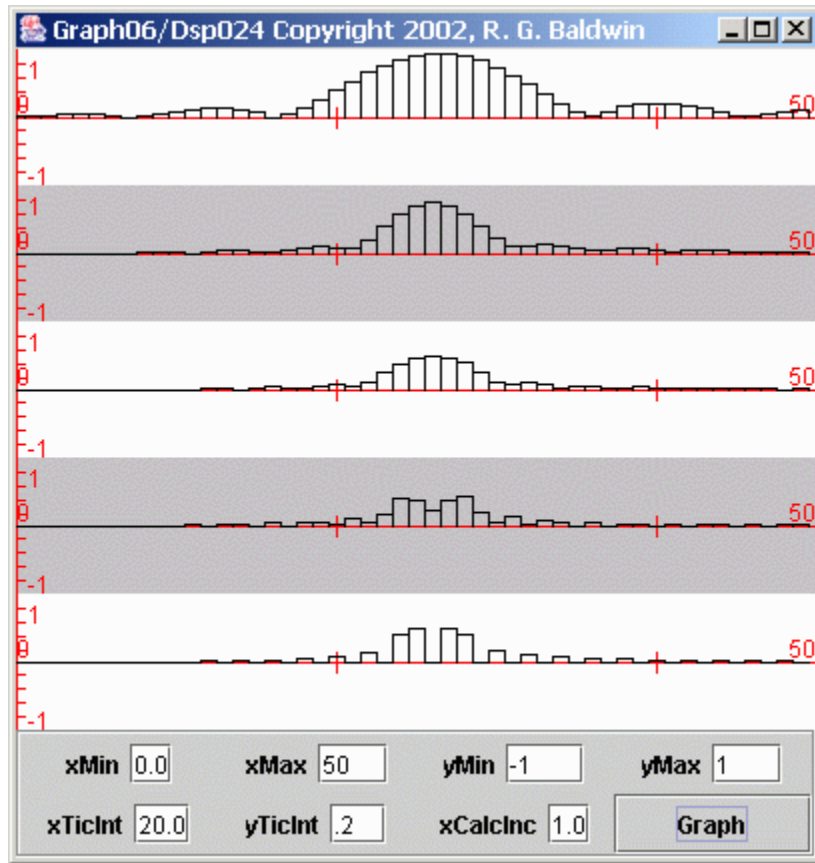
In other words, two peaks in the spectrum cannot be resolved if the frequency difference between them is less than the reciprocal of the length of the time series.

Illustration of frequency resolution

This is illustrated by the plots in [Figure 15](#). [Figure 15](#) is similar to [Figure 13](#) with one major difference. In [Figure 13](#), the frequency difference between the two sinusoids that made up each of the time series was rather large. In [Figure 15](#), the frequency difference between the two sinusoids that made up each of the time series was reduced to $1/400$, (*the reciprocal of the length of the longest time series*).

Figure 15. Illustration of frequency resolution.

Figure 15. Illustration of frequency resolution.



Each plot in [Figure 15](#) shows the first 50 points produced by performing a Fourier transform on one of the time series. In each case, the time series consisted of the sum of two sinusoids with a frequency separation of $1/400$.

A match for the frequency resolution

The length of the time series for the bottom plot was 400 samples. Thus, the separation of the two sinusoids matched the *frequency resolution* available by performing a Fourier transform on that time series.

As you can see, the two peaks in the spectrum were resolved by the bottom plot in [Figure 15](#).

Insufficient frequency resolution

The other four time series were shorter, having lengths of 80, 160, 240, and 320 samples respectively, from top to bottom.

The important thing to note in [Figure 15](#) is that the spectrum analysis performed on the 400-sample time series was successful in separating the two peaks.

However, even though the spectrum analysis on the 320-sample time series hinted at a separation of the peaks, none of the spectrum analyses on the time series that were shorter than 400 samples successfully separated the peaks.

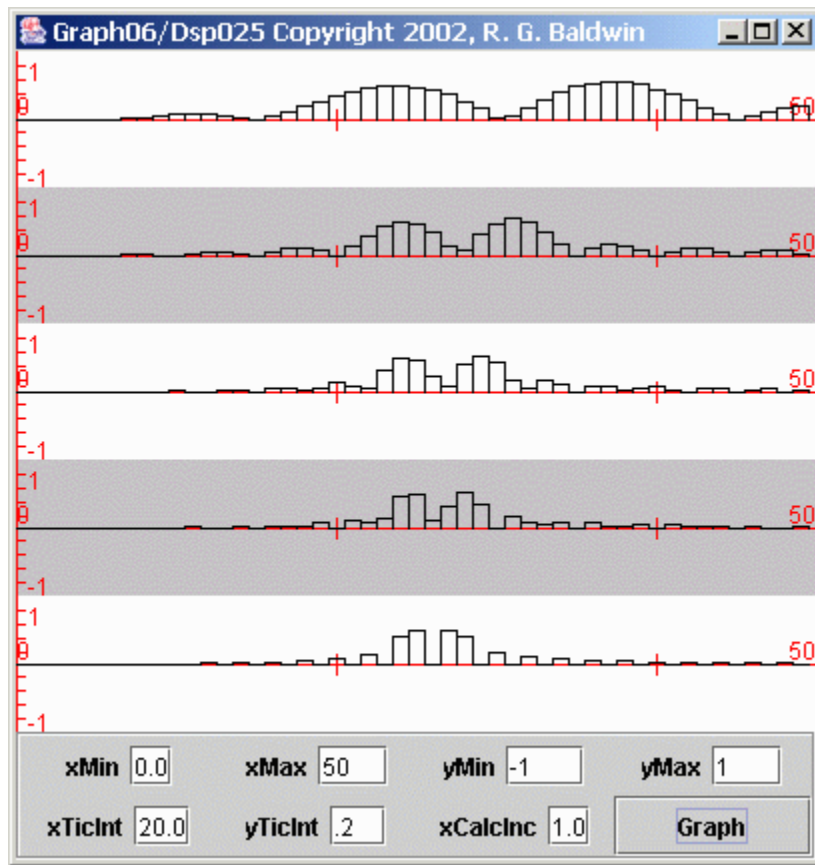
This illustrates that the frequency resolution of the Fourier transform is the reciprocal of the length of the time series.

Sufficient resolution in all five cases

I'm going to show you one more picture and then call it a wrap for this module. [Figure 16](#) is similar to [Figure 15](#) with one major difference.

Figure 16. Illustration of frequency resolution.

Figure 16. Illustration of frequency resolution.



As before, the five plots in [Figure 16](#) show the first 50 points produced by performing a Fourier transform on five different time series. Starting at the top, the lengths of the time series were 80, 160, 240, 320, and 400 samples.

Adequate frequency resolution in all cases

Also as before, each time series was the sum of two sinusoids with closely-spaced frequencies. However, in [Figure 16](#), the difference between the sinusoidal frequencies was different from one time series to the next.

In [Figure 16](#), the frequency difference for the sinusoids contained in each time series was the reciprocal of the length of that particular time series.

Therefore, the frequency difference for each case matched the frequency resolution of the Fourier transform.

The frequency of the lower-frequency peak was the same in all five cases. Therefore, this peak should line up vertically for the five plots in [Figure 16](#).

The frequency difference between the sinusoids was achieved by increasing the higher frequency by an amount equal to the reciprocal of the length of the time series.

Peaks were resolved in all five cases

If you examine [Figure 16](#), you will see that the peaks corresponding to the two sinusoids were resolved for all five time series.

As would be expected, the peaks appear to be broader for the shorter time series having the lower frequency resolution. The peaks are also separated in all five cases. However, the peaks for the lower-frequency sinusoid don't exactly line up vertically. Thus we see a small amount of measurement error in the positions of the peaks

Summary

This module has presented a pseudo-mathematical discussion of issues involving the averaging of time series, and the impact of those issues on spectrum analysis.

Those averaging issues have an impact on many other areas of DSP as well, but the detrimental effect is probably more obvious in spectrum analysis than in other areas.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Dsp00108: Digital Signal Processing (DSP) in Java, Averaging Time Series
- File: Dsp00108.htm
- Published: 12/11/02

Baldwin begins with a discussion of averaging time series, and ends with a discussion of spectral resolution, covering several related topics in between.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1468-Plotting Engineering and Scientific Data using Java
Baldwin shows you how write a generalized plotting program that can be used to plot engineering and scientific data produced by any object that implements a very simple interface.

Revised: Fri Oct 16 23:13:29 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
 - [The Graph01Demo program](#)
 - [Using the plotting program with your data](#)
 - [The Graph01 program](#)
- [Discussion and sample code](#)
 - [Testing the plotting program](#)
 - [Using the plotting program](#)
 - [Plotting your data using Graph01](#)
 - [The interface named GraphIntfc01](#)

- [The class named Graph01Demo](#)
 - [Defining data-generator classes](#)
 - [The number of functions to plot](#)
 - [The method named f1](#)
 - [The method named f2](#)
 - [The method named f3](#)
 - [The method named f4](#)
 - [The method named f5](#)
 - [The end of the class definition](#)
- [The class named Dsp002](#)
 - [A DSP example](#)
 - [Beginning of the class named Dsp002](#)
 - [Create data arrays](#)
 - [Beginning of the constructor](#)
 - [Create the convolution operator](#)
 - [Apply the convolution operator](#)
 - [Compute spectrum of each of two traces](#)
 - [The getNmbr method](#)
 - [The method named f1](#)
 - [Methods f2 through f5](#)
- [The class named Convolve01](#)
- [The discrete Fourier transform](#)
- [Two plotting programs](#)
- [The program named Graph01](#)
 - [Some general comments](#)
 - [The class named Graph01](#)
 - [Beginning of the class named GUI](#)
 - [Beginning of the constructor for the GUI class](#)
 - [Array to hold Canvas objects](#)
 - [Routine GUI construction code](#)
 - [The Canvas objects](#)
 - [More routine construction code](#)
 - [Force a repaint](#)

- [End of the constructor](#)
- [Re-plotting the data](#)
- [A new object of the target class](#)
- [The remainder of the event handler](#)
- [Beginning of the class named MyCanvas](#)
- [Beginning of the overridden paint method](#)
- [Get old coordinate values](#)
- [Plot the points](#)
- [The drawAxes method](#)
- [Drawing tic marks](#)
- [The getTheX and getTheY methods](#)
- [The test class named junk](#)
- [The program named Graph02](#)
- [Run the program](#)
- [Summary](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

Excellent language for engineering computations

Because of its platform independence, Java provides an excellent programming language for engineering and scientific computational experiments, particularly where extreme execution speed is not a requirement. Programs developed for such experiments on one platform can be successfully executed on a variety of platforms without the need to rewrite or recompile the program code.

A large Math library

Furthermore, because of its inherent simplicity, and the availability of a large Math library, Java provides an excellent programming language for engineers and scientists who want to do their own programming, but who have no desire to become programming experts. The code required to

conduct an engineering or scientific computational experiment often consists of little more than the most rudimentary application of arithmetic in loops using data stored in arrays or read from disk files.

Now for the bad news

However, there is a downside to this happy story. When doing this sort of work, it is often very important to see the results of the experiments in the form of graphs or plots. Unfortunately, the programming required to produce graphical output from simple engineering and scientific computational experiments cannot be accomplished using rudimentary programming techniques. Rather, to do that job right requires considerable expertise in Java programming.

A generalized plotting program

This module develops a generalized plotting program, which is easy to connect to other programs, (*whether they are simple or complex*) , in order to display the output from those programs in two-dimensional [Cartesian coordinates](#).. The plotting program is specifically designed to be useful to persons having *very little knowledge* of Java programming.

(Actually, the module develops two very similar plotting programs each designed to display the data in a different format.)

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Sample Display.
- [Figure 2](#). Sample Display for Same Data with Different Plotting Parameters.
- [Figure 3](#). A Digital Signal Processing (DSP) Example.
- [Figure 4](#). Graphic Display for Self-Test Class.
- [Figure 5](#). Sample output format from Graph02.

Listings

- [Listing 1](#). Source code for GraphIntfc01.java.
- [Listing 2](#). Beginning of the class named Graph01Demo.
- [Listing 3](#). The method named getNmbr.
- [Listing 4](#). The method named f1.
- [Listing 5](#). The method named f2.
- [Listing 6](#). The method named f3.
- [Listing 7](#). The method named f4.
- [Listing 8](#). The method named f5.
- [Listing 9](#). Beginning of the class named Dsp002.
- [Listing 10](#). Create data arrays.
- [Listing 11](#). Beginning of the constructor.
- [Listing 12](#). Create the convolution operator.
- [Listing 13](#). Apply the convolution operator.
- [Listing 14](#). Compute spectrum of each of two traces.
- [Listing 15](#). The getNmbr method.
- [Listing 16](#). The method named f1.
- [Listing 17](#). The class named Convolve01.
- [Listing 18](#). The discrete Fourier transform (DFT).
- [Listing 19](#). The class named Graph01.
- [Listing 20](#). Beginning of the class named GUI.
- [Listing 21](#). Beginning of the constructor for the GUI class.
- [Listing 22](#). Array to hold Canvas objects.
- [Listing 23](#). Routine GUI construction code.
- [Listing 24](#). The Canvas objects.
- [Listing 25](#). More routine construction code.
- [Listing 26](#). Force a repaint.
- [Listing 27](#). Beginning of the re-plot code.

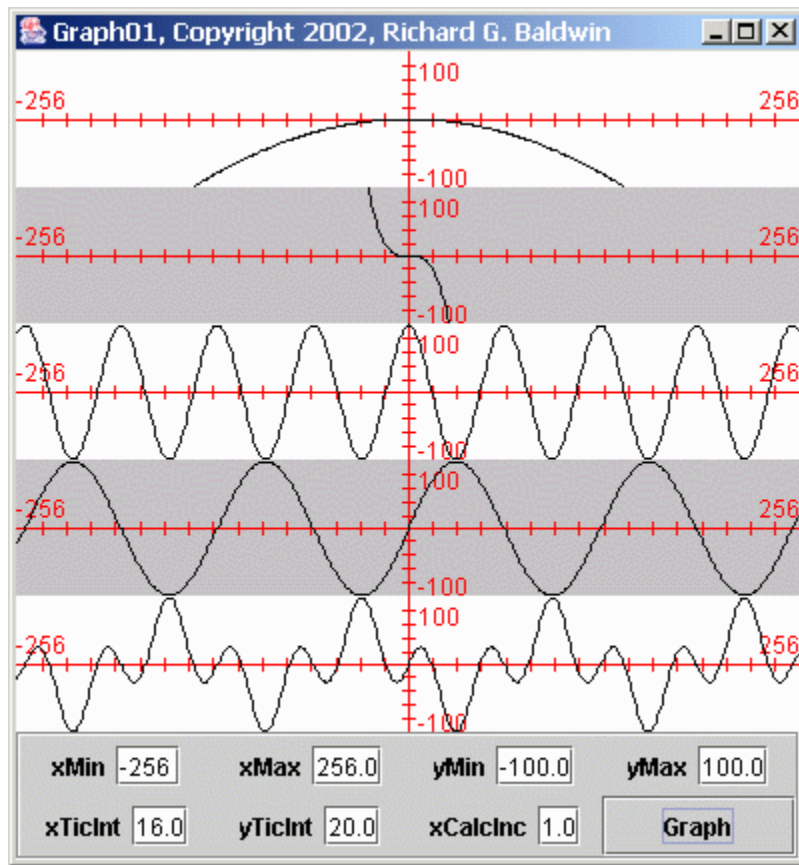
- [Listing 28](#). The remainder of the event handler.
- [Listing 29](#). Beginning of the class named MyCanvas.
- [Listing 30](#). Beginning of the overridden paint method.
- [Listing 31](#). Get old coordinate values.
- [Listing 32](#). Plot the points.
- [Listing 33](#). The drawAxes method.
- [Listing 34](#). Drawing tic marks.
- [Listing 35](#). The getTheX and getTheY methods.
- [Listing 36](#). The test class named junk.
- [Listing 37](#). Graph01Demo.java.
- [Listing 38](#). Dsp002.java,
- [Listing 39](#). Graph01.java.
- [Listing 40](#). Graph02.java.

Preview

[Figure 1](#) shows a typical display produced by one of the plotting programs that I will develop in this module. *(The other program superimposes all of the curves on the same set of axes instead of spacing them vertically as shown in [Figure 1](#).)*

Figure 1. Sample Display.

Figure 1. Sample Display.



While the plotting program itself is quite complex, the code required to produce the data to be plotted can be very simple. For example, because of the use of the Java Math library, only fourteen lines of simple Java code were required to produce the data plotted in [Figure 1](#).

The Graph01Demo program

The data displayed in [Figure 1](#) was produced by a program named **Graph01Demo**. A listing of that program is shown in [Listing 37](#) near the end of the module. I will explain that program in detail shortly.

In addition, I will provide and discuss another sample program, which produces and plots data having considerably more engineering and scientific significance than the data shown in [Figure 1](#). (*This will be a digital signal processing (DSP) example*). Even in that case, however, you will see that the program that produces the data is much less complex than the program used to plot the data.

Using the plotting program with your data

I will explain everything that you will need to know to cause the output from your own engineering and scientific programs to be displayed by the plotting program.

The Graph01 program

The graphical display of the data shown in [Figure 1](#) was produced by my generalized plotting program named **Graph01**. As you will see later, this is a long and fairly complex program.

*You will find two more variations on this program in the programs named **Graph03** and **Graph06** in the module titled [Java1482-Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#).*

A listing of the plotting program named **Graph01** is shown in [Listing 39](#) near the end of the module.

User need not understand the plotting program

Fortunately, the user of the plotting program doesn't need to understand anything about the code that makes up the plotting program. All the user

needs to understand is the interface to the program, which I will explain later.

However, for those of you who may be interested, I will also discuss and explain the plotting program later in this module.

Plotting format

As you can see in [Figure 1](#), the plotting program allows for plotting up to five independent functions stacked vertically, each with the same vertical and horizontal axes. This vertical stacking format makes it easy to compare up to five plots at the same points on the horizontal axes.

If you need more than five functions, the number of functions can easily be increased with a few minor changes to the program.

*(I will also provide, but will not discuss, another version of the program, named **Graph02**, which superimposes up to five plots on the same coordinate system. In some cases, that is a more useful form of display. You will find a complete listing of this program in [Listing 40](#) near the end of the module.)*

Plotting parameters

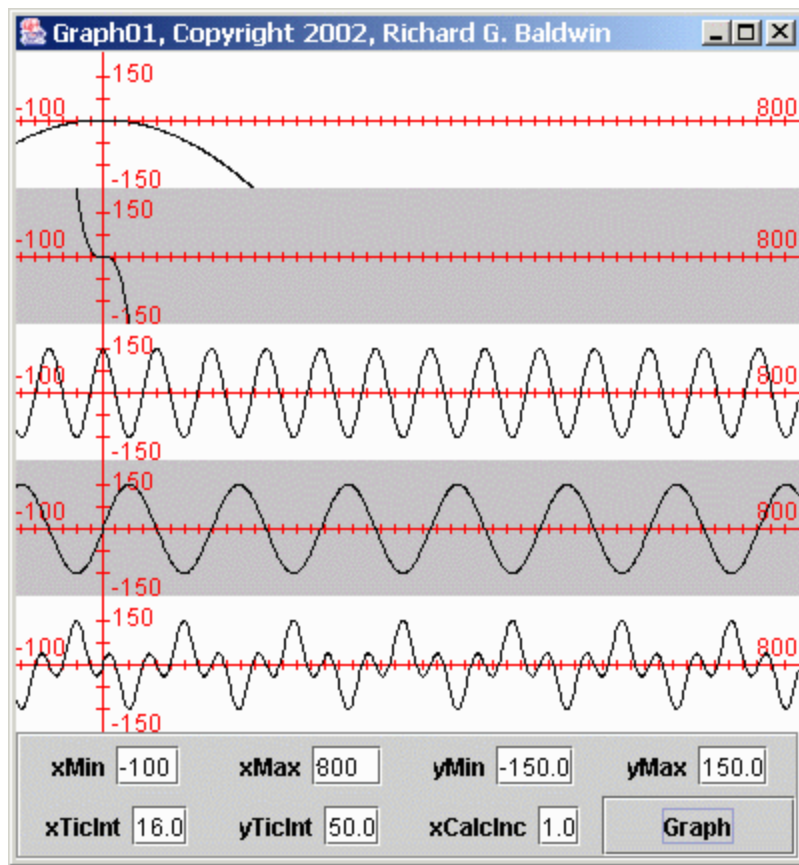
As you can also see in [Figure 1](#), a set of text fields and a button on the bottom of the frame make it possible for the user to modify the plotting parameters and to re-plot the same data with an entirely new set of plotting parameters.

(It is often true that important but subtle pieces of information can only be exposed by viewing the same data with different sets of plotting parameters.)

Same data, different parameters

[Figure 2](#) shows the same data as in [Figure 1](#), but plotted with a different set of plotting parameters.

Figure 2. Sample Display for Same Data with Different Plotting Parameters.



In the case of [Figure 2](#), the origin was moved to the left, the total expanse of the horizontal axis was increased, and the space between the tic marks on the vertical axis was increased from 20 units to 50 units.

(It will help you to see the differences if you will position two browser windows side-by-side while viewing one display in one browser window and viewing the other display in the other browser window.)

Discussion and sample code

Testing the plotting program

I am assuming that you have accomplished the minimal steps required to get the [Java Development Kit \(JDK\)](#) that is available from Oracle installed on your computer

To run the plotting program named **Graph01** in self-test mode, do the following:

- Copy the code in [Listing 39](#) into a file named **Graph01.java** .
- Copy the code in [Listing 1](#) into a file named **GraphIntfc.java** , and put that file in the same directory as the file named **Graph01.java** above.
- Compile the program named **Graph01.java** using the [Java Development Kit \(JDK\)](#). *(Note, you must be using Java Standard Edition (SE) version 1.4 or later. On the date of this module update, Java SE 8u60 is the latest released version.)*

At this point, you should be able to execute the program named **Graph01** in self-test mode by entering the following command at the command prompt in the same directory where you compiled the program:

```
java Graph01
```

If everything has been done correctly up to this point, a display similar to that shown in [Figure 4](#) should appear on your screen. *(Note that you may need to match up the parameter values in the text fields at the bottom of the*

frame and click the **Graph** button to cause the two displays to match exactly.)

Using the plotting program

To use the plotting program with your own data generator program, do the following:

- Still working in the same directory, define and compile a data generator class that implements the interface named **GraphIntfc01** , shown in [Listing 1](#).
- Start the plotting program named **Graph01** running by following the instructions that I will provide below.

Plotting your data using Graph01

Assume that your data-generator class is named **MyData** , and that you have successfully compiled it in the same directory as the compiled version of Graph01.

The next step is to enter the following command at the command prompt in the same directory. *(Note that this command differs from the command given earlier. This command provides the name of your class as a command-line argument following the name of the plotting program.)*

```
java Graph01 MyData
```

When you do this, the plotting program should start pulling the necessary data from your data-generator program and plotting that data in the format shown in [Figure 1](#).

Modifying plotting parameters

Once all the curves have been plotted, you can change any of the plotting parameter values in the text fields at the bottom of the display and press the button labeled **Graph** . When you press the button, the plotting program will re-plot your data using the new plotting parameters.

The plotting parameters

Here is the meaning of the plotting-parameter text fields shown in [Figure 1](#):

- xMin and xMax - The values of the left and right ends of all horizontal axes.
- yMin and yMax - The values of the bottom and top of the vertical axis in each plotting area. *(Note that the different plotting areas are identified by alternating white and gray backgrounds.)*
- xTicInt - The distance between tic marks on the x-axis.
- yTicInt - The distance between tic marks on the y-axis.
- xCalcInc - The distance between the points on the x-axis where values for y are computed. *(Unless your data-generator program is taking too long to run, you should probably leave this set to 1.0 in order to get the best quality plots.)*

The labels on the axes

Each x-axis has a label at the left end and the right end. Similarly, each y-axis has a label at the bottom and the top. These labels represent the values at the extreme ends of the axes. For example in [Figure 2](#), the label 800 appears at the right end of each x-axis. This is value of the x-axis where the axis intersects the border of the frame.

Keep the pixels in mind

When adjusting the plotting parameters, keep in mind that the total width of each of the plotting areas is slightly less than 400 pixels.

*(You can easily increase this to full screen width by changing one value in the **Graph01** program and recompiling the program. However, I had to keep it narrow in order to publish the images in this publication format.)*

While you can theoretically make the horizontal expanse of the x-axes as wide as you wish, because of the pixel limitation, you cannot see details that require a resolution greater than the number of pixels along the x-axis. *(This might be a good reason for you to modify the **Graph01** program as described above).*

The interface named **GraphIntfc01**

Regardless of the simplicity or complexity of your data-generator program, there are only two requirements for your program to operate successfully with the plotting program named **Graph01** :

1. It must implement the interface named **GraphIntfc01** .
2. It must have a constructor that doesn't require any parameters *(the default constructor will satisfy that requirement if you don't need another constructor)* .

Implementing **GraphIntfc01**

All that is required to implement the interface is to define a class that provides a concrete definition for each of the six methods declared in [Listing 1](#).

Listing 1. Source code for GraphIntfc01.java.

```
public interface GraphIntfc01{
    public int getNmbr();
    public double f1(double x);
    public double f2(double x);
    public double f3(double x);
    public double f4(double x);
    public double f5(double x);
} //end GraphIntfc01
```

The method named `getNmbr`

On several occasions in this module, I have stated that the plotting program can plot up to five functions. However, it doesn't have to plot all five functions. The plotting program can be used (*without modification*) to plot any number of functions from one to five.

The method named **getNmbr**, that you must define in your data-generator program, must return an integer value between 1 and 5 that specifies the number of functions to be plotted. The plotting program uses that value to divide the total plotting surface into the specified number of plotting areas, and plots each of the functions named **f1** through **fn** in one of those plotting areas.

The methods named `f1`, `f2`, `f3`, `f4`, and `f5`

As you can see in [Listing 1](#), each of these methods receives a **double** value as an incoming parameter and returns a **double** value. In essence, each of these methods receives a value for **x** and returns the corresponding value for **y**.

One plotting area per method

Each of these methods provides the data to be plotted in one plotting area. The method named **f1** provides the data for the top plotting area, the method named **f2** provides the data for the first plotting area down from the top, and so forth.

*(For example, if the **getNmbr** method returns a value of 4, the method named **f5** will never be called. If **getNmbr** returns 5, the method named **f5** will be called to provide the data for the bottom plotting area.)*

How does it work?

Each plotting area contains a horizontal axis. The plotting program moves across the horizontal axis in each plotting area one step at a time (*moving in incremental steps equal to the plotting parameter named **xCalcInc***).

At each step along the way, the plotting program calls the method associated with that plotting area, (**f1** , **f2** , etc.) , passing the horizontal position as a parameter to the method.

The value returned by the method is assumed to be the vertical value associated with that horizontal position, and that is the vertical value that is plotted for that horizontal position.

Doesn't know and doesn't care

The plotting program doesn't know, and doesn't care how the method decides on the value to return for each value that it receives as an incoming parameter. The plotting program simply calls the methods to get the data, and then plots the data.

Computed "on the fly"

For example, the returned values could be computed and returned "on the fly" as is the case in the sample program named **Graph01Demo** , which we will look at shortly.

Returned from an array

On the other hand, the values could have been computed earlier and saved in an array, as will be the case in the sample program named **Dsp002** example that we will look at later.

From a disk file, a database, the internet, etc.

The returned values could be read from a disk file, obtained from a database on another computer, or obtained from any other source such as another computer on the internet. All that matters is that when the plotting program named **Graph01** calls one of the five methods named **f1** through **f5** , passing a **double** value as a parameter, it expects to receive a **double** value as a return value, and it will plot the value that it receives.

It is up to you

It is up to you, the author of the data-generator program, to decide how you will implement the methods named **f1** through **f5** . In some cases, the implementation may be simple. In other cases, the implementation may be more complex. The first case that we will examine, **Graph01Demo** , is very simple. A subsequent case, **Dsp002** , is not so simple.

The class named **Graph01Demo**

Although this is a very simple class definition, I am going to break it down and discuss it in fragments in order to help you focus your attention on the

important points. A complete listing of the class definition is shown in [Listing 37](#) near the end of the module.

Defining data-generator classes

This class named **Graph01Demo** is used to demonstrate how to write data-generator classes that will operate successfully with the program named **Graph01** .

([Figure 1](#) shows the display of the data produced by this class. You might want to refer to that figure while examining the code in this class.)

[Listing 2](#) shows the beginning of the class definition, which names the class **Graph01Demo** , and specifies that the class implements the interface named **GraphIntfc01** .

Listing 2. Beginning of the class named Graph01Demo.

```
class Graph01Demo implements GraphIntfc01{
```

The number of functions to plot

[Listing 3](#) shows the entire listing of the method named **getNmbr** .

Listing 3. The method named getNmbr.

```
public int getNmbr(){  
    return 5;  
} //end getNmbr
```

Recall from the earlier discussion that the method named **getNmbr** must return an integer value between 1 and 5, which tells the plotting program named **Graph01** how many functions to plot.

This demonstration plots all five functions, as shown in [Figure 1](#), so this method returns the value 5.

The method named f1

[Listing 4](#) shows the entire method named **f1**. The output from this method is plotted in the topmost plotting area of the display in [Figure 1](#). (*This is the area at the top with the white background.*)

Listing 4. The method named f1.

```
public double f1(double x){  
    return -(x*x)/200.0;  
} //end f1
```


This method receives an incoming parameter known locally as **x** .
(Computations are performed on the fly by all five data-generator methods defined in this class.) The method computes and returns the negative square of the incoming parameter (divided by 200) . This produces the inverted bowl shape at the top of [Figure 1](#).

The method named f2

The curve plotted in the top-most plotting area with the gray background in [Figure 1](#) is produced by the method named **f2** , which is shown in its entirety in [Listing 5](#).

Listing 5. The method named f2.

```
public double f2(double x){  
    return -(x*x*x)/200.0;  
} //end f2
```

As before, this function receives an incoming parameter known locally as **x** . The function computes and returns the negative cube of the incoming parameter (divided by 200) . This produces the curve shown in the top-most gray area of [Figure 1](#).

The method named f3

The method named **f3** , shown in [Listing 6](#), produces the curve shown in the center plotting area with the white background in [Figure 1](#).

Listing 6. The method named f3.

```
public double f3(double x){  
    return 100*Math.cos(x/10.0);  
} //end f3
```

This is a simple cosine curve, which is computed on the fly. Each time the method is called, the incoming parameter named **x** is used to calculate the cosine of an angle in radians given by one-tenth the value of **x** . The cosine of that angle is multiplied by 100 and returned.

*(Note that the cosine of the angle is computed using a static method of the standard Java class named **Math** . This is the class that contains the Java math library.)*

The method named f4

The curve shown in the bottom-most gray plotting area of [Figure 1](#) is produced by the method named **f4** , shown in [Listing 7](#).

Listing 7. The method named f4.

Listing 7. The method named f4.

```
public double f4(double x){  
    return 100*Math.sin(x/20.0);  
} //end f4
```

The body of the method named **f4** is similar to the body of the method named **f3** , except that **f4** computes and returns sine values instead of cosine values. Also, the value of **x** is used differently so that the period of the curve produced by **f4** is twice the period of the curve produced by **f3** .

The method named f5

Finally, the bottom white plotting area in [Figure 1](#) shows the output produced by the method named **f5** , which is shown in [Listing 8](#).

Listing 8. The method named f5.

```
public double f5(double x){  
    return 100*  
(Math.sin(x/20.0)*Math.cos(x/10.0));  
} //end f5
```

This method computes and returns the product of sine and cosine functions identical to those discussed above.

The end of the class definition

[Listing 8](#) also shows the closing curly brace that signifies the end of the class definition for the class named **Graph01Demo** .

That's really all that you need to know to be able to make effective use of the generalized plotting program named **Graph01** . If you can define the methods named **f1** through **f5** , which will return the required values for your computational experiment, then you can make use of this program to plot your data.

The class named Dsp002

Lest you go away believing that this is all too trivial to be interesting, I am going to show you another example that is far from trivial. In the next example, I will demonstrate two of the most important operations in the field commonly referred to as digital signal processing, or DSP for short.

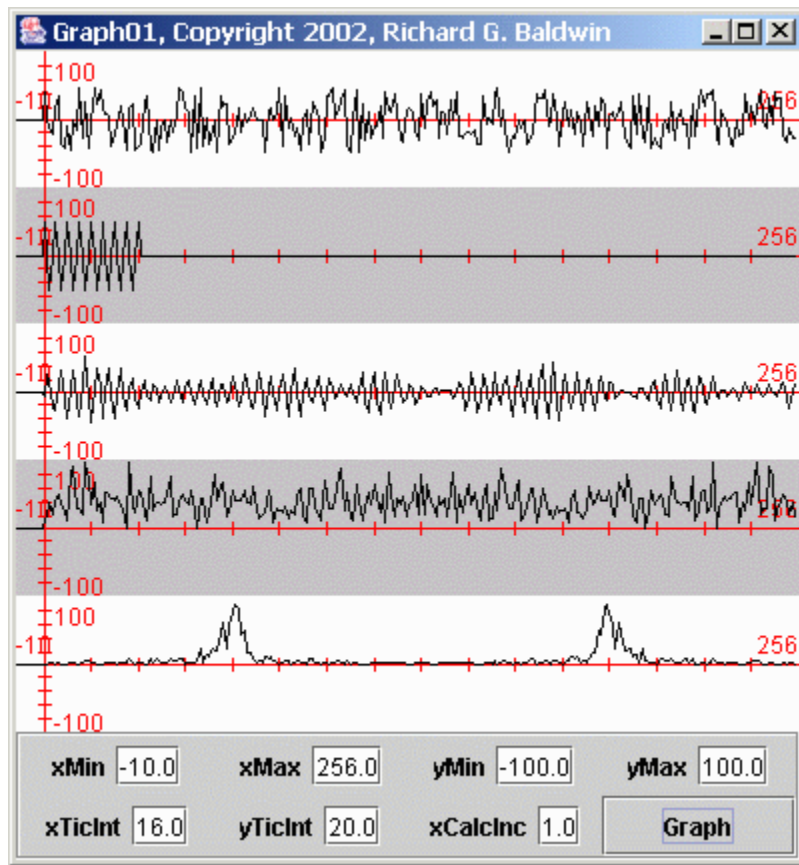
Because many of you are unlikely to be familiar with the techniques and terminology involved, the discussion will of necessity be fairly shallow. However, I do want to show at least one example of how you can perform substantive computational experiments using this approach.

A DSP example

A DSP example involving convolution filtering and spectral analysis is shown in [Figure 3](#).

Figure 3. A Digital Signal Processing (DSP) Example.

Figure 3. A Digital Signal Processing (DSP) Example.



In the field of DSP, the five individual plots shown in the plotting areas of [Figure 3](#) are commonly referred to as traces. I will use that terminology in this discussion.

White random noise

The top trace in the area with the white background shows about 256 samples of white random noise. When we get to the code, we will see that this data was created using a Java pseudo-random number generator.

A convolution filter

The second trace from the top shows a 33-point narrow-band convolution filter, which is simply a chunk taken out of a sinusoid whose frequency is one-fourth the sampling frequency. In other words, the sinusoid is represented by four samples per cycle.

The convolution filter output

The middle trace shows the result of applying the narrow-band convolution filter to the white noise. The output from the convolution process was amplified to bring it back into an appropriate amplitude range for visual analysis.

If you compare the middle trace with the top trace, you will notice that much of the high-frequency energy and much of the low-frequency energy has been removed. Most of the energy in the middle trace appears to be about the same frequency as the design frequency of the convolution filter (*which is what we would expect*) .

Time-domain vs. frequency-domain

The top three traces represent information in the time domain. The bottom two traces represent information in the frequency domain.

(Think of the frequency domain as the information that is visible on many audio systems, consisting of parallel vertical bars with lights that dance up and down. These lights are often associated with a device referred to as a frequency equalizer. When the music contains a lot of drums, or other sounds at the bass end, the lights at the low (usually left) end of the frequency spectrum are very active. When the music contains a lot of symbols, or sounds at the treble end, the lights at the high (right) end of the

frequency spectrum are very active. That is a form of real-time spectrum analysis.)

Frequency spectrum analysis

The two bottom traces in [Figure 3](#) result from performing frequency spectrum analysis on the top trace and the middle trace respectively.

The white noise spectrum

The trace in the gray area immediately below the center is an estimate of the spectral distribution of the white noise in the top trace. The spectrum analysis was performed across the frequency range from zero frequency to the sampling frequency.

While not perfectly flat, as would be the case for perfectly white noise, you can see that the energy appears to be distributed across that entire range.

(If we wanted to improve our estimate, we could capture and analyze a much longer sample of the white noise.)

If you examine this trace carefully, you might notice that there is a point of near symmetry in the middle. The values that you see above that point (*the folding frequency*) are a mirror-image of the values that you see below that point. (*I will have more to say about this later.*)

The filtered noise spectrum

The bottom trace shows an estimate of the spectral distribution of the filtered noise in the center trace. Again, the spectrum analysis was

performed across the frequency range from zero frequency to the sampling frequency. Again also, there is a symmetry point in the middle with everything to the right of that point being a mirror image of everything to the left of that point.

Two spectral peaks are visible

Unlike the spectral analysis of the white noise, this spectral analysis shows two obvious peaks. One peak appears at one-fourth the sampling frequency, and the other peak appears at three-fourths the sampling frequency.

In other words, as we concluded from examining the center trace, the filtering process removed much of the energy above and below the design frequency of the convolution filter.

(By changing the design frequency of the convolution filter, and repeating the process, we could move this peak up or down along the frequency axis.)

What does the symmetry mean?

Without getting into a lot of detail at this point, the point of symmetry that I identified above is known as the Nyquist folding frequency. *(See the earlier module titled [Dsp00104-Sampled Time Series](#).)*

In order to be able to identify the frequency of a sine wave, you must have at least two samples per cycle of the sine wave. The Nyquist folding frequency is the frequency at which you have exactly two samples per cycle.

As the frequency of the sine wave continues to increase beyond that point, without a corresponding change in the sampling frequency, it is impossible

to determine from the samples so obtained whether the frequency is increasing or decreasing.

An ambiguity in the spectrum analysis

As a result, the spectrum analysis process was unable to determine if the peak in the frequency spectrum was below or above the folding frequency. Thus, the bottom trace in [Figure 3](#) shows two peaks which are mirror images of one another with the folding frequency being half way between the two peaks.

(As a practical matter, when doing spectrum analysis, there is no point in computing the values above the folding frequency. I did that here just to illustrate that there is a folding frequency, which is equal to one-half the sampling frequency.)

Let's see some code

The class used to produce the data displayed in [Figure 3](#) is named **Dsp002** . A complete listing of this class definition is shown in [Listing 38](#) near the end of the module.

I will break this class down into fragments and briefly discuss it to show how you can define significant classes and easily connect them to the generalized plotting program named **Graph01** .

As before, having compiled the class named **Dsp002** , you would exercise it by entering the following at a command prompt:

```
java Graph01 Dsp002
```

Different from the previous example class

This class differs from the class named **Graph01Demo** in one very significant way. In that class, all the values returned by the methods named **f1** through **f5** were computed *on the fly* as the methods were called.

In this new class named **Dsp002** , all the data is generated and stored in array objects when an object of the class named **Dsp002** is instantiated. When the methods named **f1** through **f5** are called later, they simply retrieve the data from the array objects and return that data to the plotting program.

Basic operation of the program

As mentioned earlier, this program applies a narrow-band convolution filter to white noise, and then computes the amplitude spectrum of the filtered noise using a *Discrete Fourier Transform (DFT)* algorithm. The spectrum of the white noise is also computed. All of the processing occurs when an object of the class is instantiated, and the processed results are saved in arrays.

The input noise, the filter, the filtered output, and the two spectra are deposited in five arrays for later retrieval and display. The data in the five arrays are returned by the methods named **f1** , **f2** , **f3** , **f4** , and **f5** respectively.

The values that are returned by the methods are scaled for appropriate display in the plotting areas provided by the program named **Graph01** .

Beginning of the class named Dsp002

The code in [Listing 9](#) establishes the data lengths for the white noise, the convolution filter, the filtered output, and the spectrum.

Listing 9. Beginning of the class named Dsp002.

```
class Dsp002 implements GraphIntf01{
    int operatorLen = 33;
    int dataLen = 256+operatorLen;
    int outputLen = dataLen - operatorLen;
    int spectrumPts = outputLen;
```

Create data arrays

The code in [Listing 10](#) creates the array objects that will be used to store the data until it is retrieved by the methods named **f1** through **f5** .

Listing 10. Create data arrays.

```
double[] data = new double[dataLen];
double[] operator =
    new double[operatorLen];
double[] output =
    new double[outputLen];
double[] spectrumA =
    new double[spectrumPts];
double[] spectrumB =
    new double[spectrumPts];
```

Beginning of the constructor

Most of the hard work is done by the constructor or by methods called by the constructor.

The code in [Listing 11](#) shows the beginning of the constructor for the class. This code generates and saves the white noise in the array object named **data** .

Listing 11. Beginning of the constructor.

```
public Dsp002(){//constructor
    Random generator = new Random(
        new Date().getTime());
    for(int cnt=0;cnt < data.length;
        cnt++){
        //Get data, scale it, remove the
        // dc offset, and save it.
        data[cnt] = 100*generator.
            nextDouble()-50;
    }//end for loop
```

The random noise generator seed

Note that by virtue of the way this white noise is being generated, a different seed is passed to the constructor for the **Random** class each time an object of the **Dsp002** class is instantiated. Thus, each new object presents different random noise.

(In some cases, this may not be desirable and it may be preferable to use the same seed each time an object is instantiated.)

Create the convolution operator

The code in [Listing 12](#) creates the 33-point convolution operator, as a segment of a cosine wave, and saves it in the designated array.

Listing 12. Create the convolution operator.

```
for(int cnt = 0; cnt < operatorLen;
    cnt++){
    operator[cnt] = Math.cos(
        cnt*2*Math.PI/4);
} //end for loop
```

Note that the constant value of 4 in the denominator of the argument to the **cos** method specifies the frequency of the cosine wave relative to the sampling frequency. *(In this case, the frequency of the cosine wave is one-fourth the sampling frequency.)*

Apply the convolution operator

The code in [Listing 13](#) calls a static method named **convolve** in a class named **Convolve01** to apply the convolution operator to the white noise and

save the filtered result in the appropriate array. I will briefly discuss this method later.

Listing 13. Apply the convolution operator.

```
Convolve01.convolve(data,dataLen,  
                    operator,operatorLen,output);
```

Compute spectrum of each of two traces

The code in [Listing 14](#) calls a static method named **dft** of a class named **Dft01** twice in succession to compute the spectra for the white noise and the filtered noise, and to save those spectra in the appropriate arrays.

Listing 14. Compute spectrum of each of two traces.

```
Dft01.dft(data,spectrumPts,  
          spectrumA);  
Dft01.dft(output,spectrumPts,  
          spectrumB);  
} //end constructor
```

All results have been computed and saved

That is the end of the constructor. At this point, all the results have been computed and saved in the appropriate arrays for later retrieval by the methods named **f1** through **f5** .

When the constructor finishes execution, the new object of the class named **Dsp002** has been created and occupies memory. The array objects contained in the new object have been populated with five different types of data. That data is available to be retrieved and plotted.

The getNmbr method

The class definition for the class named **Dsp002** contains several more methods that I need to explain. For example, the **getNmbr** method for this class is exactly the same as for the class discussed earlier. As before, it returns the integer value 5, telling the plotting program that there are five plots to be generated. This method is shown in [Listing 15](#).

Listing 15. The getNmbr method.

```
public int getNmbr(){  
    return 5;  
} //end getNmbr
```

The method named f1

The method named **f1** is representative of all five of the methods named **f1** through **f5** in this class. Therefore, I will discuss only the first of the five methods in detail. The method named **f1** is shown in its entirety in [Listing 16](#).

Listing 16. The method named f1.

```
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > data.length-1){
        return 0;
    }else{
        return data[index];
    }//end else
}//end f1
```

In all five cases, the purpose of the method is to fetch and return a value from an array, where the incoming parameter will be converted to an array index.

Convert incoming parameter to an index

The incoming parameter is received as type **double** . However, an array must be indexed using type **int** . The first statement in the method calls the **round** method of the **Math** class to convert the **double** value to the nearest integer. That integer will be used as an array index.

Stay within array bounds

Following this, the method applies some logic to confirm that the index value is within the bounds of the array. If not, the method returns the value 0.

If the index is within the array bounds, the method retrieves and returns the value stored at that index location in the array.

And that's all there is to it.

Methods f2 through f5

Except for the scale factors applied to the data before returning it, the behavior of the methods named **f2** through **f5** is essentially the same as the behavior of the method named **f1** . In each case, the method retrieves, scales, and returns a value previously stored in an array. Therefore, I won't discuss these other methods. You can view them in [Listing 38](#) near the end of the module.

And that ends the definition of the class named **Dsp002** .

The class named Convolve01

The entire class named **Convolve01** is shown in [Listing 17](#) .

If you already understand convolution, you will probably find the code in this class straightforward. If not, the code will probably still be straightforward, but the reason for the code may be obscure.

Listing 17. The class named **Convolve01**.

```
class Convolve01{
    public static void convolve(
        double[] data,
        int dataLen,
        double[] operator,
        int operatorLen,
        double[] output){
        //Apply the operator to the data,
        // dealing with the index
        // reversal required by
        // convolution.
        for(int i=0;
            i < dataLen-operatorLen;i++){
            output[i] = 0;
            for(int j=operatorLen-1;j>=0;
                j--){
                output[i] +=
                    data[i+j]*operator[j];
            }//end inner loop
        }//end outer loop
    }//end convolve method
}//end Class Convolve01
```

To make a long story short, the class named **Convolve01** provides a static method named **convolve** , which applies an incoming convolution operator to an incoming set of data and deposits the filtered data in an output array whose reference is received as an incoming parameter.

This class could easily be broken out and put in a library as a stand-alone class, or the **convolve** method could be added to a class containing a variety of DSP methods.

The discrete Fourier transform (DFT)

The entire class named **Dft01** is shown in [Listing 18](#).

As with convolution, if you already understand the discrete Fourier transform, you will probably find the code in this class to be straightforward. If not, the code will probably still be straightforward, but the reasons for the code may be obscure. Since the purpose of this module is not to explain digital signal processing concepts, I won't attempt to provide a detailed explanation for the code in this method at this time.

Listing 18. The discrete Fourier transform (DFT).

Listing 18. The discrete Fourier transform (DFT).

```
class Dft01{
    public static void dft(
        double[] data,
        int dataLen,
        double[] spectrum){
        //Set the frequency increment to
        // the reciprocal of the data
        // length. This is convenience
        // only, and is not a requirement
        // of the DFT algorithm.
        double delF = 1.0/dataLen;
        //Outer loop iterates on frequency
        // values.
        for(int i=0; i < dataLen;i++){
            double freq = i*delF;
            double real = 0;
            double imag = 0;
            //Inner loop iterates on time-
            // series points.
            for(int j=0; j < dataLen; j++){
                real += data[j]*Math.cos(
                    2*Math.PI*freq*j);
                imag += data[j]*Math.sin(
                    2*Math.PI*freq*j);
                spectrum[i] = Math.sqrt(
                    real*real + imag*imag);
            }//end inner loop
        }//end outer loop
    }//end dft
}//end Dft01
```

Once again, to make a long story short, this class provides a static method named **dft** , which computes and returns the amplitude spectrum of an incoming time series.

The amplitude spectrum is computed as the square root of the sum of the squares of the real and imaginary parts.

A DFT algorithm can compute any number of points in the frequency domain. In this case, the number of points computed in the frequency domain is equal to the number of samples in the incoming time series, which is a fairly common practice.

The method deposits the frequency data in an array whose reference is received as an incoming parameter.

As with convolution, this class could easily be broken out and put in a library as a stand-alone class, or the **dft** method could be added to a class containing a variety of DSP methods.

Two plotting programs

Now that you have examined the sample data-generator programs, some of you may be interested in an explanation of the plotting program itself.

If you are interested only in how to use the plotting programs, and are not interested in the programming details of the plotting programs, skip ahead to the section titled [Run the Program](#).

If you are interested in learning how the plotting programs do what they do, keep reading.

Two very similar plotting programs are shown in the listings near the end of the module. The program named **Graph01** , shown in [Listing 39](#), can be used to plot as many as five separate functions, each in its own plotting area. Examples of the display produce by this program are shown in [Figure 1](#), [Figure 2](#), [Figure 3](#), and [Figure 4](#). I will briefly discuss this program in the paragraphs that follow.

The program named **Graph02** , shown in [Listing 40](#), can also be used to plot as many as five separate functions. In this case, however, the graphs produced by the functions are superimposed in the same plotting area. This is simply an alternative display format. I won't discuss any of the particulars of this program, but if you understand the program named **Graph01** , you will have no difficulty understanding this program named **Graph02** as well.

The program named Graph01

This program is designed to instantiate an object of a class file that implements the interface named **GraphIntfc01** , and to plot the data provided by up to five functions defined in that class file.

The methods in the class corresponding to the functions to be plotted are named **f1** , **f2** , **f3** , **f4** , and **f5** .

As you learned in the earlier discussion, the class containing the functions must also define a static method named **getNmbr** . This method takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a **NoSuchMethodException** will be thrown.

Some general comments

Separate plotting areas

The overall plotting surface is divided into the required number of equally sized plotting areas. One function is plotted on Cartesian coordinates in each plotting area.

A **noarg** constructor is required

The constructor for the class that implements **GraphIntfc01** must not require any parameters. This is because the **newInstance** method of the **Class** class is used to instantiate an object, based on a **String** provided as a command-line argument. The **newInstance** method can only create objects using a noarg constructor.

Some methods may not be called

If the **getNmbr** method returns a value less than 5, then the methods that will not be called begin with **f5** and work down toward **f1** . For example, if the value returned by **getNmbr** is 3, then the program will call the methods named **f1** , **f2** , and **f3** . While the methods named **f4** and **f5** must exist in order to satisfy the interface, they won't be called. Therefore, it doesn't matter what those methods return as long as it is type double.

The visual appearance

As shown in [Figure 1](#), the plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A Cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

The Cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

A test class

The program also compiles a test class named **junk** , which contains the five required methods plus the method named **getNmbr** . This makes it

easy to compile and test the program in a stand-alone mode.

Usage instructions

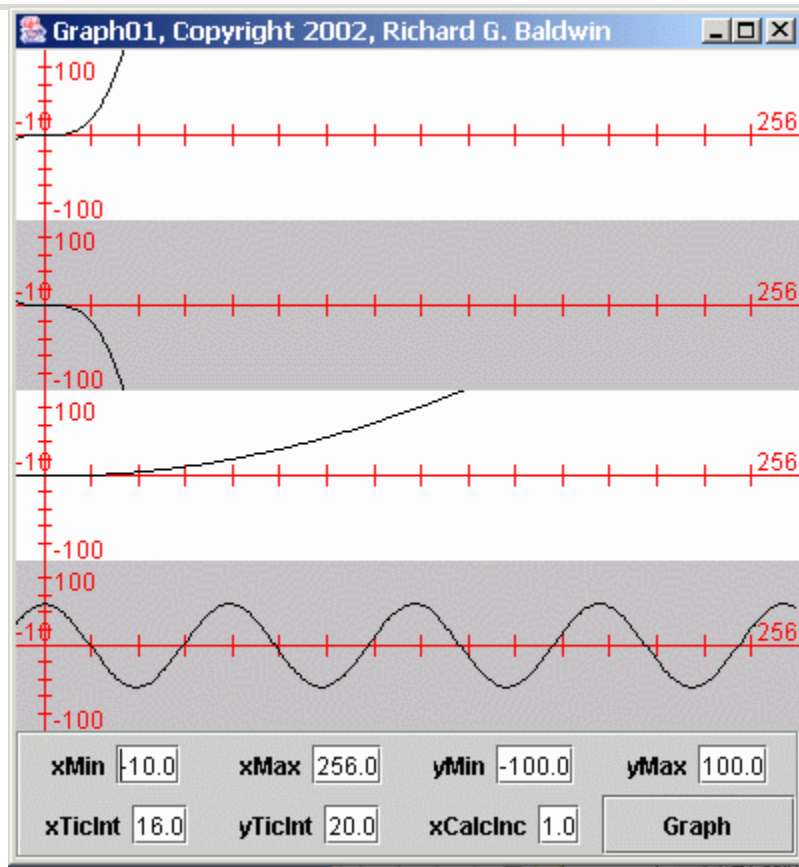
At runtime, the name of the class that implements the **GraphIntfc01** interface must be provided as a command-line argument.

If the command-line argument is missing, the program instantiates an object from the internal test class named **junk** and plots the data produced by that object. Therefore, you can test the program by running it with no command-line arguments. This will produce the display shown in [Figure 4](#).

Figure 4. Graphic Display for Self-Test Class.



Figure 4. Graphic Display for Self-Test Class.



If the command-line argument is provided, the program instantiates an object of the class whose name matches the argument, and plots the data produced by that object.

Plotting parameters

This program provides the following text fields for user input, along with a button labeled **Graph**. This allows the user to adjust the parameters and replot the graph as many times as needed with as many different plotting scales as may be needed:

- xMin = minimum x-axis value
- xMax = maximum x-axis value

- yMin = minimum y-axis value
- yMax = maximum y-axis value
- xTicInt = tic interval on x-axis
- yTicInt = tic interval on y-axis
- xCalcInc = calculation interval

The user can modify any of these parameters and then press the **Graph** button to cause the five functions to be re-plotted according to the new parameters.

A new object

Whenever the **Graph** button is pressed, the event handler for that button instantiates a new object of the class that implements the **GraphIntfc01** interface.

Depending on the nature of that class, this may be redundant in some cases. However, it is useful in those cases where it is necessary to refresh the values of instance variables defined in the class (*such as a counter, for example*) .

(I will show you how to eliminate this feature from the plotting program if you decide that it is unnecessary for your data.)

Requires Java SDK 1.4 or later

This program uses constants that were first defined in the **Color** class of v1.4.0. Therefore, the program requires v1.4.0 or later to compile and execute correctly.

Will discuss in fragments

I will discuss this program in fragments. As mentioned earlier, a complete listing of the program is provided in [Listing 39](#) near the end of the module. You should be able to copy and paste that code into your Java source file, and then compile and execute it successfully.

The class named Graph01

The entire class, including the **main** method is shown in [Listing 19](#).

Listing 19. The class named Graph01.

Listing 19. The class named Graph01.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph01{
    public static void main(
        String[] args)
        throws NoSuchMethodException,
               ClassNotFoundException,
               InstantiationException,
               IllegalAccessException{
        if(args.length == 1){
            //pass command-line parameter
            new GUI(args[0]);
        }else{
            //no command-line parameter given
            new GUI(null);
        }//end else
    }// end main
}//end class Graph01 definition
```

The primary purpose of **main** method is to instantiate an object of the class named **GUI** .

In addition, the **main** method checks to see if the user provided a command-line argument, and if so, passes it along to the constructor for the **GUI** class.

Beginning of the class named GUI

The class named **GUI** begins in [Listing 20](#).

Listing 20. Beginning of the class named GUI.

```
class GUI extends JFrame
    implements ActionListener{

    //Define plotting parameters and
    // their default values.
    double xMin = 0.0;
    double xMax = 400.0;
    double yMin = -100.0;
    double yMax = 100.0;

    //Tic mark intervals
    double xTicInt = 20.0;
    double yTicInt = 20.0;

    //Tic mark lengths.  If too small
    // on x-axis, a default value is
    // used later.
    double xTicLen = (yMax-yMin)/50;
    double yTicLen = (xMax-xMin)/50;

    //Calculation interval along x-axis
    double xCalcInc = 1.0;

    //Text fields for plotting parameters
    JTextField xMinTxt =
        new JTextField("" + xMin);
    JTextField xMaxTxt =
```

Listing 20. Beginning of the class named GUI.

```
        new JTextField("" + xMax);
JTextField yMinTxt =
        new JTextField("" + yMin);
JTextField yMaxTxt =
        new JTextField("" + yMax);
JTextField xTicIntTxt =
        new JTextField("" + xTicInt);
JTextField yTicIntTxt =
        new JTextField("" + yTicInt);
JTextField xCalcIncTxt =
        new JTextField("" + xCalcInc);

//Panels to contain a label and a
// text field
JPanel pan0 = new JPanel();
JPanel pan1 = new JPanel();
JPanel pan2 = new JPanel();
JPanel pan3 = new JPanel();
JPanel pan4 = new JPanel();
JPanel pan5 = new JPanel();
JPanel pan6 = new JPanel();

//Misc instance variables
int frmWidth = 408;
int frmHeight = 430;
int width;
int height;
int number;
GraphIntfc01 data;
String args = null;

//Plots are drawn on the canvases
// in this array.
Canvas[] canvases;
```

The code in [Listing 20](#) declares and in some cases initializes several instance variables that are required later to support the plotting process. The comments and the names of the variables generally indicate the purpose of those variables.

Beginning of the constructor for the GUI class

The beginning of the constructor for the GUI class is shown in [Listing 21](#).

Listing 21. Beginning of the constructor for the GUI class.

Listing 21. Beginning of the constructor for the GUI class.

```
GUI(String args)throws
    NoSuchMethodException,
    ClassNotFoundException,
    InstantiationException,
    IllegalAccessException{

    if(args != null){
        //Save for use later in the
        // ActionEvent handler
        this.args = args;
        //Instantiate an object of the
        // target class using the String
        // name of the class.
        data = (GraphIntfc01)
                Class.forName(args).
                newInstance();
    }else{
        //Instantiate an object of the
        // test class named junk.
        data = new junk();
    }//end else
}
```

The main purpose of the code in [Listing 21](#) is to instantiate the object that will provide the data to be plotted. If the user provided the name of a class as a command-line argument, an attempt will be made to create a **newInstance** of that class.

(In case you are unfamiliar with this approach, this is one way to create an object of a class whose name is specified as a String at runtime.)

Otherwise, the code in [Listing 21](#) will instantiate an object of the test class named **junk** (*to be discussed later*) .

Array to hold Canvas objects

Each of the separate plotting areas in [Figure 1](#) is an object of a class that extends the **Canvas** class. The code in [Listing 22](#) calls the **getNmbr** method on the new object to determine how many functions are to be plotted. Then it creates an array object to hold the requisite number of **Canvas** objects.

Listing 22. Array to hold Canvas objects.

```
//Create array to hold correct
// number of Canvas objects.
canvases =
    new Canvas[data.getNmbr()];

//Throw exception if number of
// functions is greater than 5.
number = data.getNmbr();
if(number > 5){
    throw new NoSuchMethodException(
        "Too many functions. "
        + "Only 5 allowed.");
} //end if
```

Although the limit could easily be increased, this program is currently limited to plotting the output from five functions. The code in [Listing 22](#)

checks this limit and throws an exception if an attempt is made to plot more than five functions.

Routine GUI construction code

Although somewhat long and rather tedious, the code in [Listing 23](#) is completely straightforward. This code continues with the construction of the GUI object, creating text fields, a button, etc.

Listing 23. Routine GUI construction code.

```
//Create the control panel and
// give it a border for cosmetics.
JPanel ctlPnl = new JPanel();
ctlPnl.setLayout(//?rows x 4 cols
                 new GridLayout(0,4));
ctlPnl.setBorder(
                 new EtchedBorder());

//Button for replotting the graph
JButton graphBtn =
    new JButton("Graph");
graphBtn.addActionListener(this);

//Populate each panel with a label
// and a text field. Will place
// these panels in a grid on the
// control panel later.
pan0.add(new JLabel("xMin"));
pan0.add(xMinTxt);
```

Listing 23. Routine GUI construction code.

```
pan1.add(new JLabel("xMax"));
pan1.add(xMaxTxt);

pan2.add(new JLabel("yMin"));
pan2.add(yMinTxt);

pan3.add(new JLabel("yMax"));
pan3.add(yMaxTxt);

pan4.add(new JLabel("xTicInt"));
pan4.add(xTicIntTxt);

pan5.add(new JLabel("yTicInt"));
pan5.add(yTicIntTxt);

pan6.add(new JLabel("xCalcInc"));
pan6.add(xCalcIncTxt);

//Add the populated panels and the
// button to the control panel with
// a grid layout.
ctlPnl.add(pan0);
ctlPnl.add(pan1);
ctlPnl.add(pan2);
ctlPnl.add(pan3);
ctlPnl.add(pan4);
ctlPnl.add(pan5);
ctlPnl.add(pan6);
ctlPnl.add(graphBtn);
```

Because of the routine nature of the code in [Listing 23](#), I will let the comments suffice as an explanation.

The Canvas objects

If you refer back to [Figure 1](#), you will see that from one to five **Canvas** objects are stacked vertically in the center of a frame.

This is accomplished by placing a **JPanel** object in the center of the frame, and setting the layout manager on the **JPanel** to **GridLayout**. The grid is defined as having one column and an unspecified number of rows. Then one **Canvas** object is placed in each cell of the grid, beginning at the top and working downward from the top, until the required number of **Canvas** objects have been placed in the grid.

This is accomplished by the code in [Listing 24](#).

Listing 24. The Canvas objects.

```
//Create a panel to contain the
// Canvas objects. They will be
// displayed in a one-column grid.
JPanel canvasPanel = new JPanel();
canvasPanel.setLayout(new GridLayout(0,1));

//Create a custom Canvas object for
// each function to be plotted and
// add them to the one-column grid.
// Make background colors alternate
// between white and gray.
for(int cnt = 0; cnt < number; cnt++){
    switch(cnt){
        case 0 :
```

Listing 24. The Canvas objects.

```
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.WHITE);
        break;
    case 1 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.LIGHT_GRAY);
        break;
    case 2 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.WHITE);
        break;
    case 3 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.LIGHT_GRAY);
        break;
    case 4 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].
            setBackground(Color.WHITE);
} //end switch

//Add the object to the grid.
canvasPanel.add(canvases[cnt]);
} //end for loop
```

The code in [Listing 24](#):

- Creates the JPanel object, and sets its layout property to GridLayout.
- Creates the requisite number of objects of the MyCanvas class (which extends Canvas), setting the background colors of the panels alternately to white and gray.
- Adds the MyCanvas objects to the cells in the grid. (Note that the constructor for each MyCanvas object receives an integer that specifies its position in the stack of MyCanvas objects. We will see how that information is used later.)

More routine construction code

The code in [Listing 25](#) is simply more routine code required to:

- Finish the construction of the GUI object
- Set its location and size on the screen
- Make it visible

Once again, I will let the comments serve as the explanation for this code.

Listing 25. More routine construction code.

Listing 25. More routine construction code.

```
//Add the sub-assemblies to the
// frame. Set its location, size,
// and title, and make it visible.
getContentPane().
    add(ctlPnl,"South");
getContentPane().
    add(canvasPanel,"Center");

setBounds(0,0,frmWidth,frmHeight);
setTitle("Graph01, " +
        "Copyright 2002, " +
        "Richard G. Baldwin");
setVisible(true);

//Set to exit on X-button click
setDefaultCloseOperation(
    EXIT_ON_CLOSE);

//Get and save the size of the
// plotting surface
width = canvases[0].getWidth();
height = canvases[0].getHeight();
```

Force a repaint

As you will see later, the actual plotting behavior of this program is defined by the code in an overridden version of the **paint** method in the **MyCanvas** class. I will discuss that code in some detail later.

One way to cause the code in the overridden **paint** method to be executed is to call the **repaint** method on a reference to a **MyCanvas** object.

The code in [Listing 26](#) calls the **repaint** method on each **MyCanvas** object in sequence, to guarantee that they are properly painted when the GUI object first becomes visible.

Listing 26. Force a repaint.

```
        for(int cnt = 0; cnt < number; cnt++){
            canvases[cnt].repaint();
        }//end for loop

    }//end constructor
```

Similar code will be used again later to cause the graphs to be repainted each time the user presses the **Graph** button in the bottom right corner of [Figure 1](#).

End of the constructor

The code in [Listing 26](#) also ends the constructor for the **GUI** object. When the constructor finishes execution, the **GUI** appears on the screen with all plotting areas properly painted.

Re-plotting the data

shows the beginning of the event handler that is registered on the button to cause the functions to be re-plotted.

Beginning of the re-plot code.

```
public void actionPerformed(  
                                ActionEvent evt){  
    //Re-instantiate the object that  
    // provides the data  
    try{  
        if(args != null){  
            data = (GraphIntfc01)Class.  
                forName(args).newInstance();  
        }else{  
            data = new junk();  
        }//end else  
    }catch(Exception e){  
        //Known to be safe at this point.  
        // Otherwise would have aborted  
        // earlier.  
    }//end catch
```

The purpose of the event handler is to cause the functions to be re-plotted after the user changes the plotting parameters.

A new object of the target class

However, the code in [Listing 27](#) goes beyond that. In particular, the code in [Listing 27](#) creates a new object from which to get the data that is to be plotted.

In some cases, this may be required, depending on the nature of the class from which that object is instantiated. In other cases, it may not be necessary, and could slow down the re-plotting process.

If your class doesn't contain counters or other variables that need to be re-initialized whenever you re-plot, you could probably safely remove or disable the code in [Listing 27](#). This will make the program run faster, although you may not be able to see the difference.

The remainder of the event handler

The remaining code in the event handler is shown in [Listing 28](#).

Listing 28. The remainder of the event handler.

Listing 28. The remainder of the event handler.

```
//Set plotting parameters using
// data from the text fields.
xMin = Double.parseDouble(
    xMinTxt.getText());
xMax = Double.parseDouble(
    xMaxTxt.getText());
yMin = Double.parseDouble(
    yMinTxt.getText());
yMax = Double.parseDouble(
    yMaxTxt.getText());
xTicInt = Double.parseDouble(
    xTicIntTxt.getText());
yTicInt = Double.parseDouble(
    yTicIntTxt.getText());
xCalcInc = Double.parseDouble(
    xCalcIncTxt.getText());

//Calculate new values for the
// length of the tic marks on the
// axes.  If too small on x-axis,
// a default value is used later.
xTicLen = (yMax-yMin)/50;
yTicLen = (xMax-xMin)/50;

//Repaint the plotting areas
for(int cnt = 0;
    cnt < number; cnt++){
    canvases[cnt].repaint();
} //end for loop

} //end actionPerformed
```

This code is very straightforward. It performs the following actions:

- Get new plotting parameters from the text fields.
- Perform some calculations.
- Cause each of the **MyCanvas** objects to be repainted using the new plotting parameters.

Again, I will let the comments provide any necessary explanations.

That brings us to the most interesting part of the program, the extended **Canvas** class.

Beginning of the class named MyCanvas

The class named **MyCanvas** is an inner class of the **GUI** class, which is used to override the **paint** method in each of the plotting areas shown in [Figure 1](#).

*(Virtually all graphics operations in Java, other than those involving standard GUI components, are implemented by overriding the **paint** method on an object.)*

The beginning of this class definition is shown in [Listing 29](#).

Listing 29. Beginning of the class named MyCanvas.

Listing 29. Beginning of the class named MyCanvas.

```
class MyCanvas extends Canvas{
    int cnt;//object number
    //Factors to convert from double
    // values to integer pixel locations.
    double xScale;
    double yScale;

    MyCanvas(int cnt){//save obj number
        this.cnt = cnt;
    }//end constructor
```

Floating data vs. pixels

Most of the calculations in this program are performed on data of type **double** . However, graphics operations are ultimately performed in terms integer numbers of pixels. The code in [Listing 29](#) declares scale factors used later to convert from **double** values to **integer** pixel locations.

A simple constructor

The code in [Listing 29](#) also defines the constructor, whose only purpose is to save an integer identifying the position of this object in the vertical stack of **MyCanvas** objects.

Beginning of the overridden paint method

The beginning of the overridden paint method for the **MyCanvas** class is shown in [Listing 30](#) .

Listing 30. Beginning of the overridden paint method.

```
public void paint(Graphics g){
    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
               (int)((0-yMin)*yScale));
    drawAxes(g); //Draw the axes
    g.setColor(Color.BLACK);
}
```

The code in [Listing 30](#):

- Calculates and saves the scale factors for converting from **double** coordinate values to **integer** values in pixels.
- Moves the plotting origin to the correct location.
- Calls a method to draw the axes (*in red*) on the **MyCanvas** object.
- Sets the color to black for the remainder of the plotting activity on the object.

Get old coordinate values

The plotting process consists of drawing straight line segment between pairs of points. For each line segment, one of the points is defined by a pair of old coordinate values. The other point is defined by a pair of new coordinate values.

The code in [Listing 31](#) initializes the beginning point for the plot. The initial value for the x-coordinate is the left edge of the plotting area.

Listing 31. Get old coordinate values.

```
//Get initial data values
double xVal = xMin;
int oldX = getTheX(xVal);
int oldY = 0;
//Use the Canvas obj number to
// determine which method to
// call to get the value for y.
switch(cnt){
    case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
    case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
    case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
    case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
    case 4 :
        oldY = getTheY(data.f5(xVal));
} //end switch
```

The initial y-coordinate value

The initial value for the y-coordinate depends on which function is being plotted on the **MyCanvas** object. Recall that each **MyCanvas** object contains an instance variable that identifies its position in the vertical stack of **MyCanvas** objects. The **switch** statement in [Listing 31](#) uses that information to call one of the five methods named **f1** through **f5** . This gets

the correct value for the y-coordinate based on the value of the x-coordinate for each **MyCanvas** object.

The methods named **getTheX** and **getTheY** called by the code in [Listing 31](#) convert the coordinate values from type **double** to integer values in pixels.

The method named **getTheY** also changes the sign on the data so that positive y-values go up the screen rather than down the screen.

(By default, positive vertical coordinate values go down the screen from top to bottom in Java.)

Plot the points

The remainder of the overridden paint method is shown in [Listing 32](#).

Plot the points.

```
//Now loop and plot the points
while(xVal < xMax){
    int yVal = 0;
    //Get next data value. Use the
    // Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
        case 0 :
            yVal =
```


Plot the points.

```
                getTheY(data.f1(xVal));
            break;
        case 1 :
            yVal =
                getTheY(data.f2(xVal));
            break;
        case 2 :
            yVal =
                getTheY(data.f3(xVal));
            break;
        case 3 :
            yVal =
                getTheY(data.f4(xVal));
            break;
        case 4 :
            yVal =
                getTheY(data.f5(xVal));
    }//end switch1

    //Convert the x-value to an int
    // and draw the next line segment
    int x = getTheX(xVal);
    g.drawLine(oldX,oldY,x,yVal);

    //Increment along the x-axis
    xVal += xCalcInc;

    //Save end point to use as start
    // point for next line segment.
    oldX = x;
    oldY = yVal;
} //end while loop

} //end overridden paint method
```

The code in [Listing 32](#) is relatively straightforward. This code simply iterates from the minimum x-value to the maximum x-value, calling the appropriate method (*from f1 through f5*) to get the new y-values. In the process, it calls the **drawLine** method of the **Graphics** class to connect the points.

The drawAxes method

As it turns out, it is more difficult to draw and label the axes with tic marks than it is to plot the actual data.

The code to accomplish this is shown in [Listing 33](#).

Listing 33. The drawAxes method.

```
void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Label left x-axis and bottom
    // y-axis. These are the easy
    // ones. Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                  getTheX(xMin),
                  getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                  getTheY(yMin));
```

Listing 33. The drawAxes method.

```
//Label the right x-axis and the
// top y-axis. These are the hard
// ones because the position must
// be adjusted by the font size and
// the number of characters.
//Get the width of the string for
// right end of x-axis and the
// height of the string for top of
// y-axis
//Create a string that is an
// integer representation of the
// label for the right end of the
// x-axis. Then get a character
// array that represents the
// string.
int xMaxInt = (int)xMax;
String xMaxStr = "" + xMaxInt;
char[] array = xMaxStr.
                                toCharArray();

//Get a FontMetrics object that can
// be used to get the size of the
// string in pixels.
FontMetrics fontMetrics =
                                g.getFontMetrics();
//Get a bounding rectangle for the
// string
Rectangle2D r2d =
    fontMetrics.getStringBounds(
        array,0,array.length,g);
//Get the width and the height of
// the bounding rectangle. The
// width is the width of the label
// at the right end of the
// x-axis. The height applies to
```

Listing 33. The drawAxes method.

```
// all the labels, but is needed
// specifically for the label at
// the top end of the y-axis.
int labWidth =
    (int)(r2d.getWidth());
int labHeight =
    (int)(r2d.getHeight());

//Label the positive x-axis and the
// positive y-axis using the width
// and height from above to
// position the labels. These
// labels apply to the very ends of
// the axes at the edge of the
// plotting surface.
g.drawString("" + (int)xMax,
    getTheX(xMax)-labWidth,
    getTheY(xTicLen/2)-2);
g.drawString("" + (int)yMax,
    getTheX(yTicLen/2)+2,
    getTheY(yMax)+labHeight);

//Draw the axes
g.drawLine(getTheX(xMin),
    getTheY(0.0),
    getTheX(xMax),
    getTheY(0.0));

g.drawLine(getTheX(0.0),
    getTheY(yMin),
    getTheX(0.0),
    getTheY(yMax));

//Draw the tic marks on axes
xTics(g);
```

Listing 33. The drawAxes method.

```
yTics(g);  
} //end drawAxes
```

The code in [Listing 33](#) is fairly complex, particularly with respect to putting the labels on the ends of the axes. However, I doubt that many of you are interested in the details, so I will let the comments suffice to explain the code.

Drawing tic marks

[Listing 34](#) shows the methods called from the code in [Listing 33](#) to actually draw the tic marks on the axes.

Listing 34. Drawing tic marks.

```
void xTics(Graphics g){  
    double xDoub = 0;  
    int x = 0;  
  
    //Get the ends of the tic marks.  
    int topEnd = getTheY(xTicLen/2);  
    int bottomEnd =  
        getTheY(-xTicLen/2);  
  
    //If the vertical size of the  
    // plotting area is small, the  
    // calculated tic size may be too
```

Listing 34. Drawing tic marks.

```
// small. In that case, set it to
// 10 pixels.
if(topEnd < 5){
    topEnd = 5;
    bottomEnd = -5;
} //end if

//Loop and draw a series of short
// lines to serve as tic marks.
// Begin with the positive x-axis
// moving to the right from zero.
while(xDoub < xMax){
    x = getTheX(xDoub);
    g.drawLine(x,topEnd,x,bottomEnd);
    xDoub += xTicInt;
} //end while

//Now do the negative x-axis moving
// to the left from zero
xDoub = 0;
while(xDoub > xMin){
    x = getTheX(xDoub);
    g.drawLine(x,topEnd,x,bottomEnd);
    xDoub -= xTicInt;
} //end while

} //end xTics
//-----//

//Method to draw tic marks on y-axis
void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);
```

Listing 34. Drawing tic marks.

```
//Loop and draw a series of short
// lines to serve as tic marks.
// Begin with the positive y-axis
// moving up from zero.
while(yDoub < yMax){
    y = getTheY(yDoub);
    g.drawLine(rightEnd,y,leftEnd,y);
    yDoub += yTicInt;
} //end while

//Now do the negative y-axis moving
// down from zero.
yDoub = 0;
while(yDoub > yMin){
    y = getTheY(yDoub);
    g.drawLine(rightEnd,y,leftEnd,y);
    yDoub -= yTicInt;
} //end while

} //end yTics
```

Again, I am going to let the comments suffice to explain this code.

The `getTheX` and `getTheY` methods

As mentioned earlier, methods named **getTheX** and **getTheY** are used to convert coordinate values from type **double** to integer values in pixels. Those two methods are shown in [Listing 35](#).

Listing 35. The getTheX and getTheY methods.

```
//This method translates and scales
// a double y value to plot properly
// in the integer coordinate system.
// In addition to scaling, it causes
// the positive direction of the
// y-axis to be from bottom to top.
int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
} //end getTheY
//-----//

//This method scales a double x value
// to plot properly in the integer
// coordinate system.
int getTheX(double x){
    return (int)(x*xScale);
} //end getTheX
//-----//

} //end inner class MyCanvas
//=====//

} //end class GUI
```

[Listing 35](#) also marks the end of the inner class named **MyCanvas** and the end of the class named **GUI** .

The test class named **junk**

[Listing 36](#) defines a test class named **junk** that implements the interface named **GraphIntfc01** .

Listing 36. The test class named junk.

Listing 36. The test class named junk.

```
//Sample test class. Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
    public int getNmbr(){
        //Return number of functions to
        // process. Must not exceed 5.
        return 4;
    }//end getNmbr

    public double f1(double x){
        return (x*x*x)/200.0;
    }//end f1

    public double f2(double x){
        return -(x*x*x)/200.0;
    }//end f2

    public double f3(double x){
        return (x*x)/200.0;
    }//end f3

    public double f4(double x){
        return 50*Math.cos(x/10.0);
    }//end f4

    public double f5(double x){
        return 100*Math.sin(x/20.0);
    }//end f5

}//end sample class junk
```

This class defines the methods declared in the interface, and makes it possible to test the plotting program in a stand-alone mode without having access to another class that implements the interface.

Since I discussed the implementation of this interface in some detail earlier in the module, there should be no need for me to provide further discussion of the code in [Listing 36](#). You might note, however, that since the method named **getNmbr** returns the value 4, the method named **f5** will not be called by the plotting program.

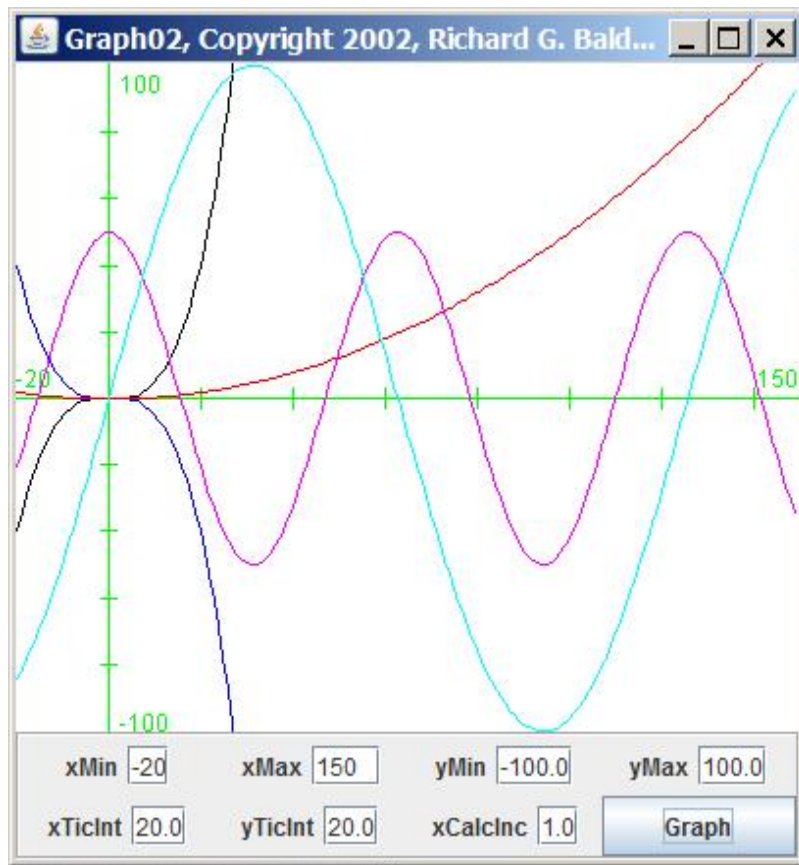
[Figure 4](#) shows the type of output produced by this self-test class.

The program named Graph02

As I mentioned earlier, the program named **Graph02**, shown in [Listing 40](#), can also be used to plot as many as five separate functions. The graphs produced by the functions are superimposed in the same plotting area. This is simply an alternative display format. A sample output from **Graph02** is shown in [Figure 5](#).

Figure 5. Sample output format from Graph02.

Figure 5. Sample output format from Graph02.



Because of the similarity of this program to **Graph01** , I won't discuss any of the particulars of this program. If you understand the program named **Graph01** , you should have no difficulty understanding the program named **Graph02** as well.

Run the program

Copy the code for the plotting program from [Listing 39](#) into a Java source file named **Graph01.java** .

Copy the code for the interface from [Listing 1](#) into a Java source file named **GraphIntfc01.java** .

Compile and run the program named **Graph01** with no command-line arguments. This should use the internal test class named **junk** discussed earlier to produce a display similar to that shown in [Figure 4](#).

Once you have the display on your screen, make changes to the plotting parameters in the text fields at the bottom and press the button labeled **Graph** . When you do, you should see the same functions being re-plotted with different plotting parameters.

Once that is successful, copy the code in [Listing 37](#) into a file named **Graph01Demo.java** . Copy the code in [Listing 38](#) into a file named **Dsp002.java** .

Compile these two files. Rerun the plotting program named **Graph01** providing **Graph01Demo** as a command-line argument. Also rerun the plotting program providing **Dsp002** as a command-line argument. This should produce displays similar to [Figure 1](#) and [Figure 3](#). *(You may need to adjust some of the plotting parameters at the bottom to make them match. Also remember that [Figure 3](#) was produced using random data so it won't be possible to match it exactly.)*

You must be running Java version 1.4 or later to successfully compile and execute this program.

Summary

I provided two generalized plotting programs in this module. One of the programs plots up to five functions in a vertical stack. The other program superimposes the plots for up to five functions on the same Cartesian coordinate system.

Each of these programs is capable of plotting the data produced by any object that implements a simple interface named **GraphIntfc01** .

I explained the interface named **GraphIntfc01** . I also explained how you can define classes of your own that implement the interface making them suitable for being plotted using either of the plotting programs.

I also provided two different sample classes that implement the interface for you to use as models as you come up to speed in defining your own classes.

Complete program listings

Complete listings of the programs discussed in this module are shown below.

Listing 37. Graph01Demo.java.

```
/* File Graph01Demo.java
Copyright 2002, R.G.Baldwin

This class is used to demonstrate how
to write data-generator classes that
will operate successfully with the
program named Graph01.

Tested using JDK 1.8 under Win 7.
*****/
class Graph01Demo
    implements GraphIntfc01{
    public int getNmbr(){
        //Return number of functions to
        // process. Must not exceed 5.
        return 5;
    }//end getNmbr

    public double f1(double x){
        //This is a simple x-squared
        // function with a negative
```

Listing 37. Graph01Demo.java.

```
// sign.  
return -(x*x)/200.0;  
} //end f1  
  
public double f2(double x){  
    //This is a simple x-cubed  
    // function  
    return -(x*x*x)/200.0;  
} //end f2  
  
public double f3(double x){  
    //This is a simple cosine  
    // function  
    return 100*Math.cos(x/10.0);  
} //end f3  
  
public double f4(double x){  
    //This is a simple sine  
    // function  
    return 100*Math.sin(x/20.0);  
} //end f4  
  
public double f5(double x){  
    //This is function which  
    // returns the product of  
    // the above sine and cosines.  
    return 100*(Math.sin(x/20.0)  
                *Math.cos(x/10.0));  
} //end f5  
  
} //end sample class Graph01Demo
```

Listing 38. Dsp002.java,

```
/* File Dsp002.java  
Copyright 2002, R.G.Baldwin
```

Note: This program requires access to the interface named GraphIntfc01.

This is a sample DSP program whose output is designed to be plotted by the programs named Graph01 and Graph02. This requires that the class implement GraphIntfc01. It also requires a noarg constructor.

This program applies a narrow-band convolution filter to white noise, and then computes the amplitude spectrum of the filtered result using a simple Discrete Fourier Transform (DFT) algorithm. The spectrum of the white noise is also computed.

The program convolves a 33-point sinusoidal convolution filter with wide-band noise, and then computes the amplitude spectrum of the raw data and the filtered result. The processing occurs when an object of the class is instantiated.

The input noise, the filter, the filtered output, and the two spectra are deposited in five arrays for later retrieval and display.

Listing 38. Dsp002.java,

The input noise, the filter, the filtered output, the spectrum of the noise, and the spectrum of the filtered result are returned by the methods named f1, f2, f3, f4, and f5 respectively.

The output values that are returned are scaled for appropriate display in the plotting areas provided by the program named Graph01.

Tested using JDK 1.8 under Win 7.

```
*****/  
import java.util.*;  
  
class Dsp002 implements GraphIntfc01{  
    //Establish data and spectrum  
    // lengths.  
    int operatorLen = 33;  
    int dataLen = 256+operatorLen;  
    int outputLen =  
        dataLen - operatorLen;  
    int spectrumPts = outputLen;  
  
    //Create arrays for the data and  
    // the results.  
    double[] data = new double[dataLen];  
    double[] operator =  
        new double[operatorLen];  
    double[] output =  
        new double[outputLen];  
    double[] spectrumA =  
        new double[spectrumPts];  
    double[] spectrumB =
```

Listing 38. Dsp002.java,

```
        new double[spectrumPts];

public Dsp002(){//constructor
    //Generate and save some wide-band
    // random noise.  Seed with a
    // different value each time the
    // object is constructed.
    Random generator = new Random(
        new Date().getTime());
    for(int cnt=0;cnt < data.length;
        cnt++){
        //Get data, scale it, remove the
        // dc offset, and save it.
        data[cnt] = 100*generator.
            nextDouble()-50;
    }//end for loop

    //Create a convolution operator and
    // save it in the array.
    for(int cnt = 0; cnt < operatorLen;
        cnt++){
        //Note, the value of the
        // denominator in the argument
        // to the cos method specifies
        // the frequency relative to the
        // sampling frequency.
        operator[cnt] = Math.cos(
            cnt*2*Math.PI/4);
    }//end for loop

    //Apply the operator to the data
    Convolve01.convolve(data,dataLen,
        operator,operatorLen,output);

    //Compute DFT of the raw data and
```

Listing 38. Dsp002.java,

```
// save it in spectrumA array.
Dft01.dft(data,spectrumPts,
           spectrumA);

//Compute DFT of the filtered data
// and save it in spectrumB array.
Dft01.dft(output,spectrumPts,
           spectrumB);

//All of the data has now been
// produced and saved. It may be
// retrieved by invoking the
// following methods named f1
// through f5.

} //end constructor

//-----//
//The following six methods are
// required by the interface named
// GraphIntfc01.
public int getNmbr(){
    //Return number of functions to
    // process. Must not exceed 5.
    return 5;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    //This version of this method
    // returns the random noise data.
    // Be careful to stay within the
    // array bounds.
    if(index < 0 ||
        index > data.length-1){
        return 0;
    }
}
```

Listing 38. Dsp002.java,

```
    }else{
        return data[index];
    }//end else
}//end f1
//-----//
public double f2(double x){
    //Return the convolution operator
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > operator.length-1){
        return 0;
    }else{
        //Scale for good visibility in
        // the plot
        return operator[index] * 50;
    }//end else
}//end f2
//-----//
public double f3(double x){
    //Return filtered output
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > output.length-1){
        return 0;
    }else{
        //Scale to approx same p-p as
        // input data
        return output[index]/6;
    }//end else
}//end f3
//-----//
public double f4(double x){
    //Return spectrum of raw data
    int index = (int)Math.round(x);
    if(index < 0 ||
```

Listing 38. Dsp002.java,

```
        index > spectrumA.length-1){
            return 0;
        }else{
            //Scale for good visibility in
            // the plot.
            return spectrumA[index]/10;
        }//end else
    }//end f4
    //-----//
    public double f5(double x){
        //Return the spectrum of the
        // filtered data.
        int index = (int)Math.round(x);
        if(index < 0 ||
            index > spectrumB.length-1){
            return 0;
        }else{
            //Scale for good visibility in
            // the plot.
            return spectrumB[index]/100;
        }//end else
    }//end f5

}//end sample class Dsp002
//=====//

//This class provides a static method
// named convolve, which applies an
// incoming convolution operator to
// an incoming set of data and deposits
// the filtered data in an output
// array whose reference is received
// as an incoming parameter.
//This class could easily be broken out
// and put in a library as a stand-
```

Listing 38. Dsp002.java,

```
// alone class, or the convolve method
// could be added to a class containing
// a variety of DSP methods.
class Convolve01{
    public static void convolve(
        double[] data,
        int dataLen,
        double[] operator,
        int operatorLen,
        double[] output){
        //Apply the operator to the data,
        // dealing with the index
        // reversal required by
        // convolution.
        for(int i=0;
            i < dataLen-operatorLen;i++){
            output[i] = 0;
            for(int j=operatorLen-1;j>=0;
                j--){
                output[i] +=
                    data[i+j]*operator[j];
            }//end inner loop
        }//end outer loop
    }//end convolve method
}//end Class Convolve01
//=====//

//This class provides a static method
// named dft, which computes and
// returns the amplitude spectrum of
// an incoming time series. The
// amplitude spectrum is computed as
// the square root of the sum of the
// squares of the real and imaginary
// parts.
```

Listing 38. Dsp002.java,

```
//Returns a number of points in the
// frequency domain equal to the number
// of samples in the incoming time
// series. Deposits the frequency
// data in an array whose reference is
// received as an incoming parameter.
//This class could easily be broken out
// and put in a library as a stand-
// alone class, or the dft method
// could be added to a class containing
// a variety of DSP methods.
class Dft01{
    public static void dft(
        double[] data,
        int dataLen,
        double[] spectrum){
        //Set the frequency increment to
        // the reciprocal of the data
        // length. This is convenience
        // only, and is not a requirement
        // of the DFT algorithm.
        double delF = 1.0/dataLen;
        //Outer loop iterates on frequency
        // values.
        for(int i=0; i < dataLen;i++){
            double freq = i*delF;
            double real = 0;
            double imag = 0;
            //Inner loop iterates on time-
            // series points.
            for(int j=0; j < dataLen; j++){
                real += data[j]*Math.cos(
                    2*Math.PI*freq*j);
                imag += data[j]*Math.sin(
                    2*Math.PI*freq*j);
            }
        }
    }
}
```

Listing 38. Dsp002.java,

```
        spectrum[i] = Math.sqrt(
            real*real + imag*imag);
    }//end inner loop
} //end outer loop
} //end dft

} //end Dft01

//=====//
```

Listing 39. Graph01.java.

```
/* File Graph01.java
Copyright 2002, R.G.Baldwin
```

Note: This program requires access to the interface named GraphIntfc01.

This is a plotting program. It is designed to access a class file, which implements GraphIntfc01, and to plot up to five functions defined in that class file. The plotting surface is divided into the required number of equally sized plotting areas, and one function is plotted on Cartesian coordinates in each area.

Listing 39. Graph01.java.

The methods corresponding to the functions are named f1, f2, f3, f4, and f5.

The class containing the functions must also define a static method named `getNmbr()`, which takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a `NoSuchMethodException` will be thrown.

Note that the constructor for the class that implements `GraphIntfc01` must not require any parameters due to the use of the `newInstance` method of the `Class` class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with f5 and work down toward f1. For example, if the number of functions is 3, then the program will expect to call methods named f1, f2, and f3. It is OK for the absent methods to be defined in the class. They simply won't be invoked.

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A Cartesian coordinate system with axes,

Listing 39. Graph01.java.

tic marks, and labels is drawn in red in each plotting area.

The Cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named junk and plots the data provided by that class. Thus, you can test the program by running it with no command-line parameter.

This program provides the following text fields for user input, along with a button labeled Graph. This allows the user to adjust the parameters and replot the graph as many times with as many plotting scales as needed:

Listing 39. Graph01.java.

```
xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval
```

The user can modify any of these parameters and then click the Graph button to cause the five functions to be re-plotted according to the new parameters.

Whenever the Graph button is clicked, the event handler instantiates a new object of the class that implements the GraphIntfc01 interface. Depending on the nature of that class, this may be redundant in some cases. However, it is useful in those cases where it is necessary to refresh the values of instance variables defined in the class (such as a counter, for example).

Tested using JDK 1.8 under Win 7.

This program uses constants that were first defined in the Color class of v1.4.0. Therefore, the program requires v1.4.0 or later to compile and run correctly.

```
*****/
```

Listing 39. Graph01.java.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph01{
    public static void main(
        String[] args)
        throws NoSuchMethodException,
               ClassNotFoundException,
               InstantiationException,
               IllegalAccessException{
        if(args.length == 1){
            //pass command-line parameter
            new GUI(args[0]);
        }else{
            //no command-line parameter given
            new GUI(null);
        }//end else
    }// end main
}//end class Graph01 definition
//=====//

class GUI extends JFrame
    implements ActionListener{

    //Define plotting parameters and
    // their default values.
    double xMin = 0.0;
    double xMax = 400.0;
    double yMin = -100.0;
    double yMax = 100.0;

    //Tic mark intervals
```

Listing 39. Graph01.java.

```
double xTicInt = 20.0;
double yTicInt = 20.0;

//Tic mark lengths.  If too small
// on x-axis, a default value is
// used later.
double xTicLen = (yMax-yMin)/50;
double yTicLen = (xMax-xMin)/50;

//Calculation interval along x-axis
double xCalcInc = 1.0;

//Text fields for plotting parameters
JTextField xMinTxt =
    new JTextField("" + xMin);
JTextField xMaxTxt =
    new JTextField("" + xMax);
JTextField yMinTxt =
    new JTextField("" + yMin);
JTextField yMaxTxt =
    new JTextField("" + yMax);
JTextField xTicIntTxt =
    new JTextField("" + xTicInt);
JTextField yTicIntTxt =
    new JTextField("" + yTicInt);
JTextField xCalcIncTxt =
    new JTextField("" + xCalcInc);

//Panels to contain a label and a
// text field
JPanel pan0 = new JPanel();
JPanel pan1 = new JPanel();
JPanel pan2 = new JPanel();
JPanel pan3 = new JPanel();
JPanel pan4 = new JPanel();
```

Listing 39. Graph01.java.

```
JPanel pan5 = new JPanel();
JPanel pan6 = new JPanel();

//Misc instance variables
int frmWidth = 408;
int frmHeight = 430;
int width;
int height;
int number;
GraphIntfc01 data;
String args = null;

//Plots are drawn on the canvases
// in this array.
Canvas[] canvases;

//Constructor
GUI(String args)throws
    NoSuchMethodException,
    ClassNotFoundException,
    InstantiationException,
    IllegalAccessException{

    if(args != null){
        //Save for use later in the
        // ActionEvent handler
        this.args = args;
        //Instantiate an object of the
        // target class using the String
        // name of the class.
        data = (GraphIntfc01)
            Class.forName(args).
                newInstance();
    }else{
        //Instantiate an object of the
```

Listing 39. Graph01.java.

```
// test class named junk.
data = new junk();
} //end else

//Create array to hold correct
// number of Canvas objects.
canvases =
    new Canvas[data.getNmbr()];

//Throw exception if number of
// functions is greater than 5.
number = data.getNmbr();
if(number > 5){
    throw new NoSuchMethodException(
        "Too many functions. "
        + "Only 5 allowed.");
} //end if

//Create the control panel and
// give it a border for cosmetics.
JPanel ctlPnl = new JPanel();
ctlPnl.setLayout(//?rows x 4 cols
    new GridLayout(0,4));
ctlPnl.setBorder(
    new EtchedBorder());

//Button for replotting the graph
JButton graphBtn =
    new JButton("Graph");
graphBtn.addActionListener(this);

//Populate each panel with a label
// and a text field. Will place
// these panels in a grid on the
// control panel later.
```

Listing 39. Graph01.java.

```
pan0.add(new JLabel("xMin"));
pan0.add(xMinTxt);

pan1.add(new JLabel("xMax"));
pan1.add(xMaxTxt);

pan2.add(new JLabel("yMin"));
pan2.add(yMinTxt);

pan3.add(new JLabel("yMax"));
pan3.add(yMaxTxt);

pan4.add(new JLabel("xTicInt"));
pan4.add(xTicIntTxt);

pan5.add(new JLabel("yTicInt"));
pan5.add(yTicIntTxt);

pan6.add(new JLabel("xCalcInc"));
pan6.add(xCalcIncTxt);

//Add the populated panels and the
// button to the control panel with
// a grid layout.
ctlPnl.add(pan0);
ctlPnl.add(pan1);
ctlPnl.add(pan2);
ctlPnl.add(pan3);
ctlPnl.add(pan4);
ctlPnl.add(pan5);
ctlPnl.add(pan6);
ctlPnl.add(graphBtn);

//Create a panel to contain the
// Canvas objects. They will be
```


Listing 39. Graph01.java.

```
// displayed in a one-column grid.
JPanel canvasPanel = new JPanel();
canvasPanel.setLayout(//?rows,1 col
                      new GridLayout(0,1));

//Create a custom Canvas object for
// each function to be plotted and
// add them to the one-column grid.
// Make background colors alternate
// between white and gray.
for(int cnt = 0;
    cnt < number; cnt++){
    switch(cnt){
        case 0 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.WHITE);

            break;
        case 1 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.LIGHT_GRAY);

            break;
        case 2 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.WHITE);

            break;
        case 3 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
```

Listing 39. Graph01.java.

```
                                Color.LIGHT_GRAY);
        break;
    case 4 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].
            setBackground(Color.WHITE);
    }//end switch
    //Add the object to the grid.
    canvasPanel.add(canvases[cnt]);
} //end for loop

//Add the sub-assemblies to the
// frame. Set its location, size,
// and title, and make it visible.
getContentPane().
    add(ctlPnl,"South");
getContentPane().
    add(canvasPanel,"Center");

setBounds(0,0,frmWidth,frmHeight);
setTitle("Graph01, " +
        "Copyright 2002, " +
        "Richard G. Baldwin");
setVisible(true);

//Set to exit on X-button click
setDefaultCloseOperation(
    EXIT_ON_CLOSE);

//Get and save the size of the
// plotting surface
width = canvases[0].getWidth();
height = canvases[0].getHeight();
```

Listing 39. Graph01.java.

```
//Guarantee a repaint on startup.
for(int cnt = 0;
    cnt < number; cnt++){
    canvases[cnt].repaint();
} //end for loop

} //end constructor
//-----//

//This event handler is registered
// on the JButton to cause the
// functions to be replotted.
public void actionPerformed(
   (ActionEvent evt){
    //Re-instantiate the object that
    // provides the data
    try{
        if(args != null){
            data = (GraphIntfc01)Class.
                forName(args).newInstance();
        }else{
            data = new junk();
        } //end else
    }catch(Exception e){
        //Known to be safe at this point.
        // Otherwise would have aborted
        // earlier.
    } //end catch

    //Set plotting parameters using
    // data from the text fields.
    xMin = Double.parseDouble(
        xMinTxt.getText());
    xMax = Double.parseDouble(
        xMaxTxt.getText());
```

Listing 39. Graph01.java.

```
yMin = Double.parseDouble(
    yMinTxt.getText());
yMax = Double.parseDouble(
    yMaxTxt.getText());
xTicInt = Double.parseDouble(
    xTicIntTxt.getText());
yTicInt = Double.parseDouble(
    yTicIntTxt.getText());
xCalcInc = Double.parseDouble(
    xCalcIncTxt.getText());

//Calculate new values for the
// length of the tic marks on the
// axes.  If too small on x-axis,
// a default value is used later.
xTicLen = (yMax-yMin)/50;
yTicLen = (xMax-xMin)/50;

//Repaint the plotting areas
for(int cnt = 0;
    cnt < number; cnt++){
    canvases[cnt].repaint();
} //end for loop

} //end actionPerformed
//-----//

//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
    int cnt;//object number
    //Factors to convert from double
    // values to integer pixel locations.
```

Listing 39. Graph01.java.

```
double xScale;
double yScale;

MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
}//end constructor

//Override the paint method
public void paint(Graphics g){
    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);

    //Get initial data values
    double xVal = xMin;
    int oldX = getTheX(xVal);
    int oldY = 0;
    //Use the Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
        case 0 :
            oldY = getTheY(data.f1(xVal));
            break;
        case 1 :
            oldY = getTheY(data.f2(xVal));
            break;
        case 2 :
```

Listing 39. Graph01.java.

```
        oldY = getTheY(data.f3(xVal));
        break;
    case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
    case 4 :
        oldY = getTheY(data.f5(xVal));
} //end switch

//Now loop and plot the points
while(xVal < xMax){
    int yVal = 0;
    //Get next data value. Use the
    // Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
        case 0 :
            yVal =
                getTheY(data.f1(xVal));
            break;
        case 1 :
            yVal =
                getTheY(data.f2(xVal));
            break;
        case 2 :
            yVal =
                getTheY(data.f3(xVal));
            break;
        case 3 :
            yVal =
                getTheY(data.f4(xVal));
            break;
        case 4 :
            yVal =
```

Listing 39. Graph01.java.

```
                getTheY(data.f5(xVal));
        }//end switch1

        //Convert the x-value to an int
        // and draw the next line segment
        int x = getTheX(xVal);
        g.drawLine(oldX,oldY,x,yVal);

        //Increment along the x-axis
        xVal += xCalcInc;

        //Save end point to use as start
        // point for next line segment.
        oldX = x;
        oldY = yVal;
    }//end while loop

} //end overridden paint method
//-----//

//Method to draw axes with tic marks
// and labels in the color RED
void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Label left x-axis and bottom
    // y-axis. These are the easy
    // ones. Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                  getTheX(xMin),
                  getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
```

Listing 39. Graph01.java.

```
getTheY(yMin));

//Label the right x-axis and the
// top y-axis. These are the hard
// ones because the position must
// be adjusted by the font size and
// the number of characters.
//Get the width of the string for
// right end of x-axis and the
// height of the string for top of
// y-axis
//Create a string that is an
// integer representation of the
// label for the right end of the
// x-axis. Then get a character
// array that represents the
// string.
int xMaxInt = (int)xMax;
String xMaxStr = "" + xMaxInt;
char[] array = xMaxStr.
    toCharArray();

//Get a FontMetrics object that can
// be used to get the size of the
// string in pixels.
FontMetrics fontMetrics =
    g.getFontMetrics();
//Get a bounding rectangle for the
// string
Rectangle2D r2d =
    fontMetrics.getStringBounds(
        array,0,array.length,g);
//Get the width and the height of
// the bounding rectangle. The
// width is the width of the label
```


Listing 39. Graph01.java.

```
// at the right end of the
// x-axis. The height applies to
// all the labels, but is needed
// specifically for the label at
// the top end of the y-axis.
int labWidth =
    (int)(r2d.getWidth());
int labHeight =
    (int)(r2d.getHeight());

//Label the positive x-axis and the
// positive y-axis using the width
// and height from above to
// position the labels. These
// labels apply to the very ends of
// the axes at the edge of the
// plotting surface.
g.drawString("" + (int)xMax,
    getTheX(xMax)-labWidth,
    getTheY(xTicLen/2)-2);
g.drawString("" + (int)yMax,
    getTheX(yTicLen/2)+2,
    getTheY(yMax)+labHeight);

//Draw the axes
g.drawLine(getTheX(xMin),
    getTheY(0.0),
    getTheX(xMax),
    getTheY(0.0));

g.drawLine(getTheX(0.0),
    getTheY(yMin),
    getTheX(0.0),
    getTheY(yMax));
```

Listing 39. Graph01.java.

```
//Draw the tic marks on axes
xTics(g);
yTics(g);
} //end drawAxes

//-----//

//Method to draw tic marks on x-axis
void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
        getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small. In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
        topEnd = 5;
        bottomEnd = -5;
    } //end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
        x = getTheX(xDoub);
        g.drawLine(x, topEnd, x, bottomEnd);
        xDoub += xTicInt;
    }
}
```

Listing 39. Graph01.java.

```
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
        x = getTheX(xDoub);
        g.drawLine(x,topEnd,x,bottomEnd);
        xDoub -= xTicInt;
    }//end while

} //end xTics
//-----//

//Method to draw tic marks on y-axis
void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
        y = getTheY(yDoub);
        g.drawLine(rightEnd,y,leftEnd,y);
        yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
```

Listing 39. Graph01.java.

```
        y = getTheY(yDoub);
        g.drawLine(rightEnd,y,leftEnd,y);
        yDoub -= yTicInt;
    }//end while

} //end yTics

//-----//

//This method translates and scales
// a double y value to plot properly
// in the integer coordinate system.
// In addition to scaling, it causes
// the positive direction of the
// y-axis to be from bottom to top.
int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
} //end getTheY
//-----//

//This method scales a double x value
// to plot properly in the integer
// coordinate system.
int getTheX(double x){
    return (int)(x*xScale);
} //end getTheX
//-----//

} //end inner class MyCanvas
//=====//

} //end class GUI
//=====//
```

Listing 39. Graph01.java.

```
//Sample test class. Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntf01{
    public int getNmbr(){
        //Return number of functions to
        // process. Must not exceed 5.
        return 4;
    }//end getNmbr

    public double f1(double x){
        return (x*x*x)/200.0;
    }//end f1

    public double f2(double x){
        return -(x*x*x)/200.0;
    }//end f2

    public double f3(double x){
        return (x*x)/200.0;
    }//end f3

    public double f4(double x){
        return 50*Math.cos(x/10.0);
    }//end f4

    public double f5(double x){
        return 100*Math.sin(x/20.0);
    }//end f5

}//end sample class junk
```

Listing 40. Graph02.java.

```
/* File Graph02.java  
Copyright 2002, R.G.Baldwin
```

Note: This program requires access to the interface named GraphIntfc01.

This is a modified version of the program named Graph01. That program plots up to five separate curves in separate plotting areas. This program superimposes up to five separate curves in different colors in the same plotting area.

This is a plotting program. It is designed to access a class file, which implements GraphIntfc01, and to plot up to five functions defined in that class file.

The methods corresponding to the functions are named f1, f2, f3, f4, and f5.

The class containing the functions must also define a static method named getNmbr(), which takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a `NoSuchMethodException` will be thrown.

Note that the constructor for the class that implements GraphIntfc01 must not

Listing 40. Graph02.java.

require any parameters due to the use of the newInstance method of the Class class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with f5 and work down toward f1. For example, if the number of functions is 3, then the program will expect to call methods named f1, f2, and f3. It is OK for the absent methods to be defined in the class. They simply won't be invoked.

Each curve is plotted in a different color. The correspondence between colors and function calls is as follows:

f1: BLACK
f2: BLUE
f3: RED
f4: MAGENTA
f5: CYAN

A Cartesian coordinate system with axes, tic marks, and labels is drawn in green.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample

Listing 40. Graph02.java.

class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfrc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named junk and plots the data provided by that class. Thus, you can test the program by running it with no command-line parameter.

This program provides the following text fields for user input, along with a button labeled Graph. This allows the user to adjust the parameters and replot the graph as many times with as many plotting scales as needed:

```
xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval
```

The user can modify any of these parameters and then click the Graph button to cause the five functions to be re-plotted according to the

Listing 40. Graph02.java.

new parameters.

Whenever the Graph button is clicked, the event handler instantiates a new object of the class that implements the GraphIntfc01 interface. Depending on the nature of that class, this may be redundant in some cases. However, it is useful in those cases where it is necessary to refresh the values of instance variables defined in the class (such as a counter, for example).

Tested using JDK 1.8 under Win 7.

This program uses constants that were first defined in the Color class of v1.4.0. Therefore, the program requires v1.4.0 or later to compile and run correctly.

```
*****/
```

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;
```

```
class Graph02{
    public static void main(
        String[] args)
        throws NoSuchMethodException,
               ClassNotFoundException,
               InstantiationException,
               IllegalAccessException{
```

Listing 40. Graph02.java.

```
    if(args.length == 1){
        //pass command-line parameter
        new GUI(args[0]);
    }else{
        //no command-line parameter given
        new GUI(null);
    }//end else
} // end main
} //end class Graph02 definition
//=====//

class GUI extends JFrame
    implements ActionListener{

    //Define plotting parameters and
    // their default values.
    double xMin = 0.0;
    double xMax = 400.0;
    double yMin = -100.0;
    double yMax = 100.0;

    //Tic mark intervals
    double xTicInt = 20.0;
    double yTicInt = 20.0;

    //Tic mark lengths.  If too small
    // on x-axis, a default value is
    // used later.
    double xTicLen = (yMax-yMin)/50;
    double yTicLen = (xMax-xMin)/50;

    //Calculation interval along x-axis
    double xCalcInc = 1.0;

    //Text fields for plotting parameters
```

Listing 40. Graph02.java.

```

    JTextField xMinTxt =
        new JTextField("" + xMin);
    JTextField xMaxTxt =
        new JTextField("" + xMax);
    JTextField yMinTxt =
        new JTextField("" + yMin);
    JTextField yMaxTxt =
        new JTextField("" + yMax);
    JTextField xTicIntTxt =
        new JTextField("" + xTicInt);
    JTextField yTicIntTxt =
        new JTextField("" + yTicInt);
    JTextField xCalcIncTxt =
        new JTextField("" + xCalcInc);

    //Panels to contain a label and a
    // text field
    JPanel pan0 = new JPanel();
    JPanel pan1 = new JPanel();
    JPanel pan2 = new JPanel();
    JPanel pan3 = new JPanel();
    JPanel pan4 = new JPanel();
    JPanel pan5 = new JPanel();
    JPanel pan6 = new JPanel();

    //Misc instance variables
    int frmWidth = 408;
    int frmHeight = 430;
    int width;
    int height;
    int number;
    GraphIntfc01 data;
    String args = null;

    //Plots are drawn on theCanvas

```

Listing 40. Graph02.java.

```
Canvas theCanvas;

//Constructor
GUI(String args)throws
        NoSuchMethodException,
        ClassNotFoundException,
        InstantiationException,
        IllegalAccessException{

    if(args != null){
        //Save for use later in the
        // ActionEvent handler
        this.args = args;
        //Instantiate an object of the
        // target class using the String
        // name of the class.
        data = (GraphIntfc01)
                Class.forName(args).
                    newInstance();
    }else{
        //Instantiate an object of the
        // test class named junk.
        data = new junk();
    }//end else

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
        throw new NoSuchMethodException(
            "Too many functions. "
            + "Only 5 allowed.");
    }//end if

    //Create the control panel and
```

Listing 40. Graph02.java.

```
// give it a border for cosmetics.
JPanel ctlPnl = new JPanel();
ctlPnl.setLayout(//?rows x 4 cols
                 new GridLayout(0,4));
ctlPnl.setBorder(
                 new EtchedBorder());

//Button for replotting the graph
JButton graphBtn =
    new JButton("Graph");
graphBtn.addActionListener(this);

//Populate each panel with a label
// and a text field. Will place
// these panels in a grid on the
// control panel later.
pan0.add(new JLabel("xMin"));
pan0.add(xMinTxt);

pan1.add(new JLabel("xMax"));
pan1.add(xMaxTxt);

pan2.add(new JLabel("yMin"));
pan2.add(yMinTxt);

pan3.add(new JLabel("yMax"));
pan3.add(yMaxTxt);

pan4.add(new JLabel("xTicInt"));
pan4.add(xTicIntTxt);

pan5.add(new JLabel("yTicInt"));
pan5.add(yTicIntTxt);

pan6.add(new JLabel("xCalcInc"));
```

Listing 40. Graph02.java.

```
pan6.add(xCalcIncTxt);

//Add the populated panels and the
// button to the control panel with
// a grid layout.
ctlPnl.add(pan0);
ctlPnl.add(pan1);
ctlPnl.add(pan2);
ctlPnl.add(pan3);
ctlPnl.add(pan4);
ctlPnl.add(pan5);
ctlPnl.add(pan6);
ctlPnl.add(graphBtn);

//Create a custom Canvas object for
// all functions to be plotted on.
theCanvas = new MyCanvas();
theCanvas.setBackground(
                                Color.WHITE);

//Add the sub-assemblies to the
// frame. Set its location, size,
// and title, and make it visible.
getContentPane().add(
                                ctlPnl,"South");
getContentPane().add(
                                theCanvas,"Center");

setBounds(0,0,frmWidth,frmHeight);
setTitle("Graph02, " +
                                "Copyright 2002, " +
                                "Richard G. Baldwin");
setVisible(true);

//Set to exit on X-button click
```

Listing 40. Graph02.java.

```
        setDefaultCloseOperation(  
                                EXIT_ON_CLOSE);  
  
        //Get and save the size of the  
        // plotting surface  
        width = theCanvas.getWidth();  
        height = theCanvas.getHeight();  
  
        //Guarantee a repaint on startup.  
        theCanvas.repaint();  
  
    }//end constructor  
    //-----//  
  
    //This event handler is registered  
    // on the JButton to cause the  
    // functions to be replotted.  
    public void actionPerformed(  
                                ActionEvent evt){  
        //Re-instantiate the object that  
        // provides the data  
        try{  
            if(args != null){  
                data = (GraphIntfc01)Class.  
                    forName(args).newInstance();  
            }else{  
                data = new junk();  
            }//end else  
        }catch(Exception e){  
            //Known to be safe at this point.  
            // Otherwise would have aborted  
            // earlier.  
        }//end catch  
  
        //Set plotting parameters using
```

Listing 40. Graph02.java.

```
// data from the text fields.
xMin = Double.parseDouble(
    xMinTxt.getText());
xMax = Double.parseDouble(
    xMaxTxt.getText());
yMin = Double.parseDouble(
    yMinTxt.getText());
yMax = Double.parseDouble(
    yMaxTxt.getText());
xTicInt = Double.parseDouble(
    xTicIntTxt.getText());
yTicInt = Double.parseDouble(
    yTicIntTxt.getText());
xCalcInc = Double.parseDouble(
    xCalcIncTxt.getText());

//Calculate new values for the
// length of the tic marks on the
// axes. If too small on x-axis,
// a default value is used later.
xTicLen = (yMax-yMin)/50;
yTicLen = (xMax-xMin)/50;

//Repaint the plotting area
theCanvas.repaint();

} //end actionPerformed
//-----//

//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
    //Factors to convert from double
```


Listing 40. Graph02.java.

```
// values to integer pixel locations.
double xScale;
double yScale;

//Override the paint method
public void paint(Graphics g){
    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g); //Draw the axes

    //Draw each curve in a different
    // color.
    for(int cnt=0; cnt < number;
        cnt++){

        //Get initial data values
        double xVal = xMin;
        int oldX = getTheX(xVal);
        int oldY = 0;
        //Use the curve number to
        // determine which method to
        // invoke to get the value for y.
        switch(cnt){
            case 0 :
                oldY= getTheY(data.f1(xVal));
                g.setColor(Color.BLACK);
                break;
            case 1 :
                oldY= getTheY(data.f2(xVal));
```

Listing 40. Graph02.java.

```

        g.setColor(Color.BLUE);
        break;
    case 2 :
        oldY= getTheY(data.f3(xVal));
        g.setColor(Color.RED);
        break;
    case 3 :
        oldY= getTheY(data.f4(xVal));
        g.setColor(Color.MAGENTA);
        break;
    case 4 :
        oldY= getTheY(data.f5(xVal));
        g.setColor(Color.CYAN);
} //end switch

//Now loop and plot the points
while(xVal < xMax){
    int yVal = 0;
    //Get next data value. Use the
    // curve number to determine
    // which method to invoke to
    // get the value for y.
    switch(cnt){
        case 0 :
            yVal = getTheY(
                        data.f1(xVal));
            break;
        case 1 :
            yVal = getTheY(
                        data.f2(xVal));
            break;
        case 2 :
            yVal = getTheY(
                        data.f3(xVal));
            break;
    }
    g.fillRect(xVal, yVal, 1, 1);
    xVal++;
    cnt++;
}

```

Listing 40. Graph02.java.

```
        case 3 :
            yVal = getTheY(
                        data.f4(xVal));
            break;
        case 4 :
            yVal = getTheY(
                        data.f5(xVal));
    }//end switch1

    //Convert the x-value to an int
    // and draw the next line
    // segment
    int x = getTheX(xVal);
    g.drawLine(oldX,oldY,x,yVal);

    //Increment along the x-axis
    xVal += xCalcInc;

    //Save end point to use as
    // start point for next line
    // segment.
    oldX = x;
    oldY = yVal;
} //end while loop

} //end for loop

} //end overridden paint method
//-----//

//Method to draw axes with tic marks
// and labels in the color GREEN
void drawAxes(Graphics g){
    g.setColor(Color.GREEN);
```

Listing 40. Graph02.java.

```
//Label left x-axis and bottom
// y-axis. These are the easy
// ones. Separate the labels from
// the ends of the tic marks by
// two pixels.
g.drawString("" + (int)xMin,
             getTheX(xMin),
             getTheY(xTicLen/2)-2);
g.drawString("" + (int)yMin,
             getTheX(yTicLen/2)+2,
             getTheY(yMin));

//Label the right x-axis and the
// top y-axis. These are the hard
// ones because the position must
// be adjusted by the font size and
// the number of characters.
//Get the width of the string for
// right end of x-axis and the
// height of the string for top of
// y-axis
//Create a string that is an
// integer representation of the
// label for the right end of the
// x-axis. Then get a character
// array that represents the
// string.
int xMaxInt = (int)xMax;
String xMaxStr = "" + xMaxInt;
char[] array = xMaxStr.
                toCharArray();

//Get a FontMetrics object that can
// be used to get the size of the
// string in pixels.
```

Listing 40. Graph02.java.

```
FontMetrics fontMetrics =
    g.getFontMetrics();
//Get a bounding rectangle for the
// string
Rectangle2D r2d =
    fontMetrics.getStringBounds(
        array,0,array.length,g);

//Get the width and the height of
// the bounding rectangle. The
// width is the width of the label
// at the right end of the
// x-axis. The height applies to
// all the labels, but is needed
// specifically for the label at
// the top end of the y-axis.
int labWidth =
    (int)(r2d.getWidth());
int labHeight =
    (int)(r2d.getHeight());

//Label the positive x-axis and the
// positive y-axis using the width
// and height from above to
// position the labels. These
// labels apply to the very ends of
// the axes at the edge of the
// plotting surface.
g.drawString("" + (int)xMax,
    getTheX(xMax)-labWidth,
    getTheY(xTicLen/2)-2);
g.drawString("" + (int)yMax,
    getTheX(yTicLen/2)+2,
    getTheY(yMax)+labHeight);
```

Listing 40. Graph02.java.

```
//Draw the axes
g.drawLine(getTheX(xMin),
           getTheY(0.0),
           getTheX(xMax),
           getTheY(0.0));

g.drawLine(getTheX(0.0),
           getTheY(yMin),
           getTheX(0.0),
           getTheY(yMax));

//Draw the tic marks on axes
xTics(g);
yTics(g);
} //end drawAxes

//-----//

//Method to draw tic marks on x-axis
void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
        getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small. In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
        topEnd = 5;
    }
}
```

Listing 40. Graph02.java.

```
        bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
        x = getTheX(xDoub);
        g.drawLine(x,topEnd,x,bottomEnd);
        xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
        x = getTheX(xDoub);
        g.drawLine(x,topEnd,x,bottomEnd);
        xDoub -= xTicInt;
    }//end while

} //end xTics
//-----//

//Method to draw tic marks on y-axis
void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
```

Listing 40. Graph02.java.

```
// moving up from zero.
while(yDoub < yMax){
    y = getTheY(yDoub);
    g.drawLine(rightEnd,y,leftEnd,y);
    yDoub += yTicInt;
} //end while

//Now do the negative y-axis moving
// down from zero.
yDoub = 0;
while(yDoub > yMin){
    y = getTheY(yDoub);
    g.drawLine(rightEnd,y,leftEnd,y);
    yDoub -= yTicInt;
} //end while

} //end yTics

//-----//

//This method translates and scales
// a double y value to plot properly
// in the integer coordinate system.
// In addition to scaling, it causes
// the positive direction of the
// y-axis to be from bottom to top.
int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
} //end getTheY
//-----//

//This method scales a double x value
// to plot properly in the integer
```


Listing 40. Graph02.java.

```
// coordinate system.
int getTheX(double x){
    return (int)(x*xScale);
} //end getTheX
//-----//

} //end inner class MyCanvas
//=====//

} //end class GUI
//=====//

//Sample test class. Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
    public int getNmbr(){
        //Return number of functions to
        // process. Must not exceed 5.
        return 5;
    } //end getNmbr

    public double f1(double x){
        return (x*x*x)/200.0;
    } //end f1

    public double f2(double x){
        return -(x*x*x)/200.0;
    } //end f2

    public double f3(double x){
        return (x*x)/200.0;
    } //end f3

    public double f4(double x){
```

Listing 40. Graph02.java.

```
        return 50*Math.cos(x/10.0);  
    }//end f4  
  
    public double f5(double x){  
        return 100*Math.sin(x/20.0);  
    }//end f5  
  
}//end sample class junk
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1468-Plotting Engineering and Scientific Data using Java
- File: Java1468.htm
- Published: 04/17/02

Baldwin shows you how write a generalized plotting program that can be used to plot engineering and scientific data produced by any object that implements a very simple interface.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1489-Plotting 3D Surfaces using Java

Learn how to write a Java class that uses color to plot 3D surfaces in six different formats and a wide range of sizes. The class is extremely easy to use. You can incorporate the 3D plotting capability into your own programs by inserting a single statement into your programs.

Revised: Fri Oct 16 23:14:43 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General Discussion](#)
 - [Displaying 3D data can be fairly difficult](#)
 - [Different approaches are available](#)
 - [A 3D plotting program](#)
 - [Grayscale plot](#)
 - [Color Shift plot](#)
 - [Color Contour plot](#)
 - [A logarithmic conversion](#)
 - [Extremely easy to use](#)
- [Preview](#)

- Constructor
 - Parameters
 - A calibration scale
 - Normalization
- The main method
 - A 3D parabola
- The program named ImgMod29
 - The main method
 - Local variables
 - A 3D parabolic surface
 - Display six surface images
 - The ImgMod29 class
 - Establish display format for log conversion
 - Copy the input elevation data
 - Convert to log data if required
 - Normalize the surface elevation data
 - The method named scaleTheSurfaceData
 - Create an appropriate pair of Canvas objects
 - Add the Canvas objects to the Frame
 - Register an anonymous WindowListener object
 - Six different inner classes
 - The getCenter method
 - Grayscale plot format
 - The constructor
 - Overridden paint method for CanvasType0surface class
 - Instantiate a Color object
 - Set colors and draw squares
 - Draw the optional red axes
 - Beginning of the class named CanvasType0scale

- [The overridden paint method](#)
- [Color Shift plot format](#)
 - [Beginning of the class named CanvasType1surface](#)
 - [Overridden paint method](#)
 - [Set white and black for max and min values](#)
 - [Elevations other than the extreme ends](#)
 - [Processing the other three ranges](#)
 - [The class named CanvasType1scale](#)
- [Color Contour plot format](#)
 - [The color palette](#)
 - [The CanvasType2surface class](#)
 - [Set the color value](#)
 - [The class named CanvasType2scale](#)
- [Run the program](#)
- [Summary](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

In one of my earlier modules titled [Plotting Engineering and Scientific Data using Java](#), I published a generalized 2D plotting program that makes it easy to cause other programs to display their outputs in 2D [Cartesian coordinates](#). I have used that plotting program in numerous modules since I originally published it several years ago. Hopefully, some of my readers have been using it as well.

In this module, I will present and explain a 3D surface plotting program that is also very easy to use.

Numerous Java graphics libraries are available from various locations on the web. Some are of high quality, and some are not. Unfortunately, many of those libraries have a rather substantial learning curve.

The purpose of this program

It is not the purpose of the class that I will provide in this module to compete with those graphics libraries. Rather, this class is intended to make it possible for an experienced Java programmer to incorporate 3D surface plotting capability into a Java program with a learning curve of three minutes or less.

(If you are an experienced Java programmer, you can start your three-minute learning-curve clock right now. If you are not an experienced Java programmer, it may take a little longer but should still be easy. If you need to work on your Java programming skills, see [Object-Oriented Programming \(OOP\) with Java](#).)

How do you use the class?

All that's necessary to use this class to plot your own 3D surfaces is to copy and compile the source code in [Listing 29](#) near the end of this module. Then include a statement similar to the following in your program:

```
new ImgMod29(data, blockSize, true, 0);
```

The 3D surface data to be plotted

The parameter named **data** in the above statement is a reference to a 2D array of type **double** that contains the sampled elevation values of the surface to be plotted.

The granularity of the plot

The second parameter named **blockSize** specifies the size of one side of a square array of colored pixels in the final plot that will represent each elevation point on your 3D surface. Set this to 0 if you are unsure as to what size square you need.

*(If you look very carefully, you may be able to see a small white square at the center of the middle image in [Figure 1](#). This is a nine-pixel square produced by a **blockSize** value of 3.)*

The optional axes

The third parameter specifies whether or not you want to have optional axes drawn on the plot. (See [Figure 3](#) for examples of plots with and without the axes.) A parameter value of true causes the axes to be drawn. A value of false causes the axes to be omitted.

The plotting format

The fourth parameter is an integer that specifies the plotting format as follows:

- 0 - Grayscale (linear, top-left image in [Figure 2](#))
- 1 - Color Shift (linear, top-center image in [Figure 2](#))
- 2 - Color Contour (linear, top-right image in [Figure 2](#))
- 3 - Grayscale with logarithmic data conversion (bottom-left image in [Figure 2](#))
- 4 - Color Shift with logarithmic data conversion (bottom-center image in [Figure 2](#))
- 5 - Color Contour with logarithmic data conversion (bottom-right image in [Figure 2](#))

If you are unsure as to which format would be best for your application, just start with a value of 0. Then try all six formats to see which one works best for you.

Extremely simple to use

The class couldn't be simpler to use.

(Your three-minute learning curve has expired. You now know how to use the class to incorporate 3D surface plotting in your Java programs.)

Will use in subsequent modules

This 3D plotting class will be used in numerous future modules involving such complex topics as the use of the 2D Fourier Transform to process images.

If you arrived at this page seeking a free Java program for plotting your 3D surfaces, you are in luck. Just copy the source code for the class in [Listing 29](#) and feel free to use it as described above.

On the other hand, if you would like to learn how the class does what it does, and perhaps use your programming skills to improve it, keep reading. Hopefully, once you have finished the module, you will have learned quite a lot about plotting 3D surfaces using color in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). 3D views of a wave-number spectrum.
- [Figure 2](#). Sample output with logarithmic flattening.

- [Figure 3](#). Program output in self-test mode.

Listings

- [Listing 1](#). Beginning of the main method.
- [Listing 2](#). A 3D parabolic surface,
- [Listing 3](#). Display six surface images.
- [Listing 4](#). Beginning of the ImgMod29 class.
- [Listing 5](#). Establish display format for log conversion.
- [Listing 6](#). Copy the input elevation data.
- [Listing 7](#). Convert to log data if required
- [Listing 8](#). Normalize the surface elevation data.
- [Listing 9](#). The method named scaleTheSurfaceData.
- [Listing 10](#). Create an appropriate pair of Canvas objects.
- [Listing 11](#). Add the Canvas objects to the Frame.
- [Listing 12](#). Register an anonymous WindowListener object.
- [Listing 13](#). The method named getCenter.
- [Listing 14](#). Beginning of the class named CanvasType0surface.
- [Listing 15](#). Beginning of overridden paint method.
- [Listing 16](#). Instantiate a Color object.
- [Listing 17](#). Set colors and draw squares.
- [Listing 18](#). Draw the optional red axes.
- [Listing 19](#). Beginning of the class named CanvasType0scale .
- [Listing 20](#). The overridden paint method.
- [Listing 21](#). Beginning of the class named CanvasType1surface.
- [Listing 22](#). Beginning of the overridden paint method.
- [Listing 23](#). Set white and black for max and min values.
- [Listing 24](#). Process elevations from 1 to 63 inclusive.
- [Listing 25](#). Processing the other three ranges.
- [Listing 26](#). The method named getColorPalette.
- [Listing 27](#). Beginning of overridden paint method.
- [Listing 28](#). Set the color value.
- [Listing 29](#). Source code for ImgMod29.java.

General Discussion

Displaying 3D data can be fairly difficult

One of the more difficult aspects of engineering and scientific computing is displaying three-dimensional (3D) surfaces in ways that are meaningful to persons who need to view and to analyze those surfaces. The basic problem is that it is necessary to display the 3D surface on a 2D media, such as a computer screen.

(At least that was true before the advent of 3D printers. However, as of October 2015, a 3D printer is not readily available for routine use by most people.)

Therefore, some compromise is always required.

Different approaches are available

Various [approaches](#) have been devised for accomplishing this objective including:

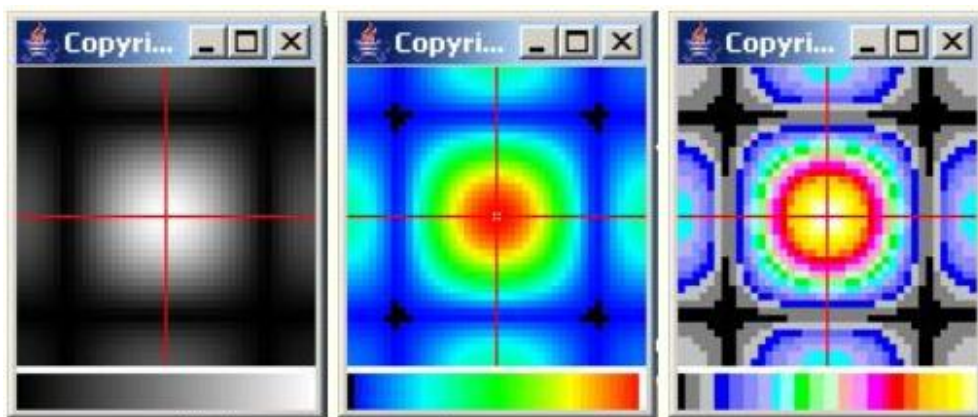
- [Grayscale plots](#)
- Color Shift plots
- Color Contour plots
- Using light and shadows to render the surface in ways that simulate a photograph
- [Labeled Numeric Contour plots](#)
- [Isometric Drawings](#)

A 3D plotting program

I will provide and explain a program in this module that makes it very easy to display a 3D surface as a Grayscale plot, a Color Shift plot, or a Color Contour plot. The program supports six different plotting formats. Three of those plotting formats are illustrated in [Figure 1](#).

(The images shown in [Figure 1](#) are different views of the wave-number spectrum resulting from performing a 2D Fourier Transform on a box in the space domain. The use of 2D Fourier Transforms will be the main topic of a future module.)

Figure 1. 3D views of a wave-number spectrum.



[Figure 1](#) shows the same 3D surface plotted using three different plotting formats. Going from left to right, [Figure 1](#) shows:

- Grayscale plot
- Color Shift plot
- Color Contour plot

Grayscale plot

The plot on the left in [Figure 1](#) is the old standby method in which the elevation of each point on the surface is represented by a shade of gray with the highest elevation being white and the lowest elevation being black.

A calibration scale

Each of the images in [Figure 1](#) shows a calibration scale immediately below the image of the surface. The calibration scale shows the color or shade of gray used to represent each elevation on the surface. The color or shade of the lowest elevation is shown on the left of the scale and the color or shade of the highest elevation is shown on the right.

For example, the scale on the Grayscale image shows a smooth gradient from black to white going from left to right. The shade of gray shown at the midpoint on the calibration scale represents the elevation that is halfway between the lowest elevation and the highest elevation.

Color Shift plot

The image in the center in [Figure 1](#) shows the same surface plotted using a smooth gradient from blue at the low end through aqua, green, and yellow to red at the high end. In addition, this plotting format sets the lowest elevation to black and the highest elevation to white so that these two elevations are obvious in the plot.

(The highest elevation was indicated by a small white square at the center in the original program output, and the lowest elevations were indicated by the black areas near the corners. However, the small white square seems to have faded away in the published version of this plot.)

More information is conveyed

This plotting format can convey a great deal more information to a human observer than the Grayscale plotting format. This is because the human eye can discern more different colors than it can discern different shades of gray.

(Many years ago, when I was in the SONAR business, the general rule of thumb was that a typical human can discern only about seven shades of gray from black to white inclusive. Obviously a typical human can discern more than seven different colors.)

The lowest elevations are obvious in the Color Shift plot

By using black to indicate the lowest elevation, *(in addition to using shades of blue to indicate low elevations)*, it is easy to determine the exact locations of the lowest elevations in the Color Shift plot in the center of [Figure 1](#). On the other hand, the locations of the lowest elevations are not discernable in the Grayscale plot at the left [Figure 1](#).

Four minor peaks are obvious in the Color Shift plot

Also, it is obvious from the Color Shift plot that the surface has four minor peaks at the edges of the plot. Although the Grayscale plot has a slight hint of those minor peaks, they are certainly not obvious.

By comparing the color at the top of the minor peaks to the color scale below the Color Shift surface, it is possible to estimate that the elevation of the minor peaks is probably somewhere between twenty-five and fifty percent of the elevation of the main central peak. It is clearly impossible to glean that kind of information from the Grayscale plot of the same surface.

The highest elevation is obvious in the Color Shift plot

Although no longer true in this reproduction of the original computer output, by using white to indicate the highest elevation *(in addition to using shades of*

red to indicate high elevations) , it is easy to determine the exact location of the highest elevation in the Color Shift plot. As explained earlier, the highest elevation was indicated by a small white square, which was on the cross hairs at the center of the original plot.

While the highest elevation is pretty well indicated in the Grayscale plot also, if the central peak were not symmetric in all four quadrants, there might be some uncertainty as to the exact location of the highest elevation.

Quantitative estimates are possible

In addition to providing a good overview of the shape of the 3D surface, the Color Shift plot also makes it possible to estimate the elevations of points on the surface in a quantitative way. For example, in addition to black and white, the yellow and aqua colors are fairly easy to identify on the surface plot and are also fairly easy to identify on the calibration scale. In addition, the yellow and aqua bands on the calibration scale are fairly narrow. By measuring the locations of these colors on the calibration scale, the elevations of the yellow and aqua areas on the surface plot can be estimated with reasonable accuracy.

The ranges of the surface elevations colored red, green, and blue can also be estimated but with less certainty.

(Because the green portion of the calibration scale is about twice as wide as the red and blue portions, the level of uncertainty when using the green calibration data to estimate the elevation of a point on the surface is about twice the level of uncertainty when using the red or blue calibration data to estimate the elevation of a point on the surface.)

Color Contour plot

The rightmost image in [Figure 1](#), which is a Color Contour plot, improves on the ability to provide good quantitative estimates of surface elevations.

If you need to read elevations off the surface plot to a high level of accuracy, the best approach is probably to use a [Labeled Numeric Contour plot](#).. However, such plots are relatively difficult to create. Also, because of the need for the labels to be large enough to read, the space required to display such a plot can sometimes be excessive.

The Color Contour plot is a reasonable compromise between a Labeled Numeric Contour plot and a Color Shift plot. This plotting format provides more accuracy in estimating surface elevations than the Color Shift plot, but doesn't require any more space to display.

Similar to a contour map

The Color Contour plot at the right in [Figure 1](#) is similar to a contour map without labels on the contours. Each color traces out a constant elevation on the surface. The elevation indicated by a given color on the 3D surface can be determined by the position of that color in the calibration scale at the bottom of the image.

(This program quantizes the range from the lowest to the highest elevation into 23 levels. Therefore, the accuracy of an elevation estimate is good to only about one 23rd of that total range. However, it would be an easy matter to increase the number of quantization levels used in this program, thereby improving the accuracy of elevation estimates.)

For example, the blue contour that surrounds the central peak traces out the shape of an elevation that is about three levels up from the lowest elevation (*as seen on the calibration scale*) . The red contour that surrounds the central peak traces out the shape of an elevation that is about five levels down from the highest elevation. The aqua at the center of each of four minor peaks establishes their peak elevation to be about thirty-five percent of the elevation of the central peak (*based on the position of aqua in the calibration scale*) .

More minor peaks

This plotting format also exposes four more minor peaks at the corners of the plot. The light gray color indicates that the level of these peaks is about two levels up from the lowest elevation. These four peaks are barely visible in the Color Shift plot in the center, and their elevation is clearly not quantifiable in that plotting format. They are not visible at all in the Grayscale plot on the left end of [Figure 1](#).

The design of this program

There are numerous options available when designing a Color Contour Plotting program. As mentioned above, the Color Contour plotting format in this program subdivides the surface into 23 elevation levels including the lowest and the highest levels. Then it represents each elevation level with a different color or shade of gray. This causes a lot of quantitative information to become available that isn't available with either of the other two formats.

(There is nothing unique about 23 elevation levels and 23 colors. It would be very easy to use many more levels and many more colors. The biggest difficulty when designing the Color Contour format is identifying a large number of colors that are clearly identifiable both on the calibration scale and on the surface plot. This raises the question as to how many unique colors a human can discern. Theoretically, a computer program such as this can generate more than sixteen million different colors. Obviously, a human cannot discern sixteen million unique colors.)

Good quantitative elevation information is available

To determine the elevation associated with a particular color, all you need to do is to locate that color on the calibration scale and determine its position relative to the colors at the ends. That will tell you the elevation associated with that color relative to the highest elevation and the lowest elevation. For

example, the color at the exact center of the calibration scale represents an elevation that is half way between the lowest elevation and the highest elevation.

(With this program, there are no absolute elevations. Rather, the calibration scale indicates each elevation level as a percentage of the difference between the lowest and the highest elevations.)

The elevations of the minor peaks

Once again, this image shows that the elevations of the four minor peaks on the edges match the color aqua on the calibration scale. Judging from the position of the color aqua on the calibration scale, the elevation of each of the four minor peaks is about thirty-five percent of the elevation of the major peak in the center.

This information is clearly not available from the Grayscale plot. It is also not available with this degree of accuracy from the Color Shift plot.

(All that we can tell from the Color Shift plot is that the minor peaks are some shade of green, which represents a rather large range of possible elevations.)

The lowest elevations

The Color Shift plot does a better job of identifying the locations of the lowest elevations than does the Color Contour plot. This is because the color black was dedicated to that purpose in the Color Shift plot, but was used to represent a range of elevations in the Color Contour plot.

(The color black could also be dedicated to identifying the lowest elevation in the design of the Color Contour plot, in which case, both schemes would be equal in this regard.)

As we learned earlier, the lowest elevation occurs at four different points on the surface, and those points are near the corners.

The elevation of the valleys

Both plots show that while the valleys between the central and minor peaks are very deep, they aren't quite as deep as the lowest elevation. They are blue in the Color Shift plot and gray in the Color Contour plot. Because the gray color represents a somewhat smaller elevation range in the Color Contour plot than the blue represents in the Color Shift plot, the elevation of the valley is defined more accurately in the Color Contour plot.

A logarithmic conversion

Sometimes when plotting data, it is useful to plot the logarithm of the data values instead of the raw values.

(For many years, engineers have plotted data on graph paper referred to either as semi-log paper or log-log paper. Each type of graph paper has advantages and disadvantages relative to the other type and also has advantages and disadvantages relative to linear graph paper. This program provides a capability that is analogous to the use of semi-log paper but in a 3D sense.)

Flattens the plot

The use of semi-log paper has the effect of flattening the plot in the 2D case, or flattening the surface in the 3D case. The semi-log approach tends to pull the structure of the low-level values up so that they can be better observed. The logarithm of the low elevations is closer to the maximum elevation than is the raw value of the low elevations.

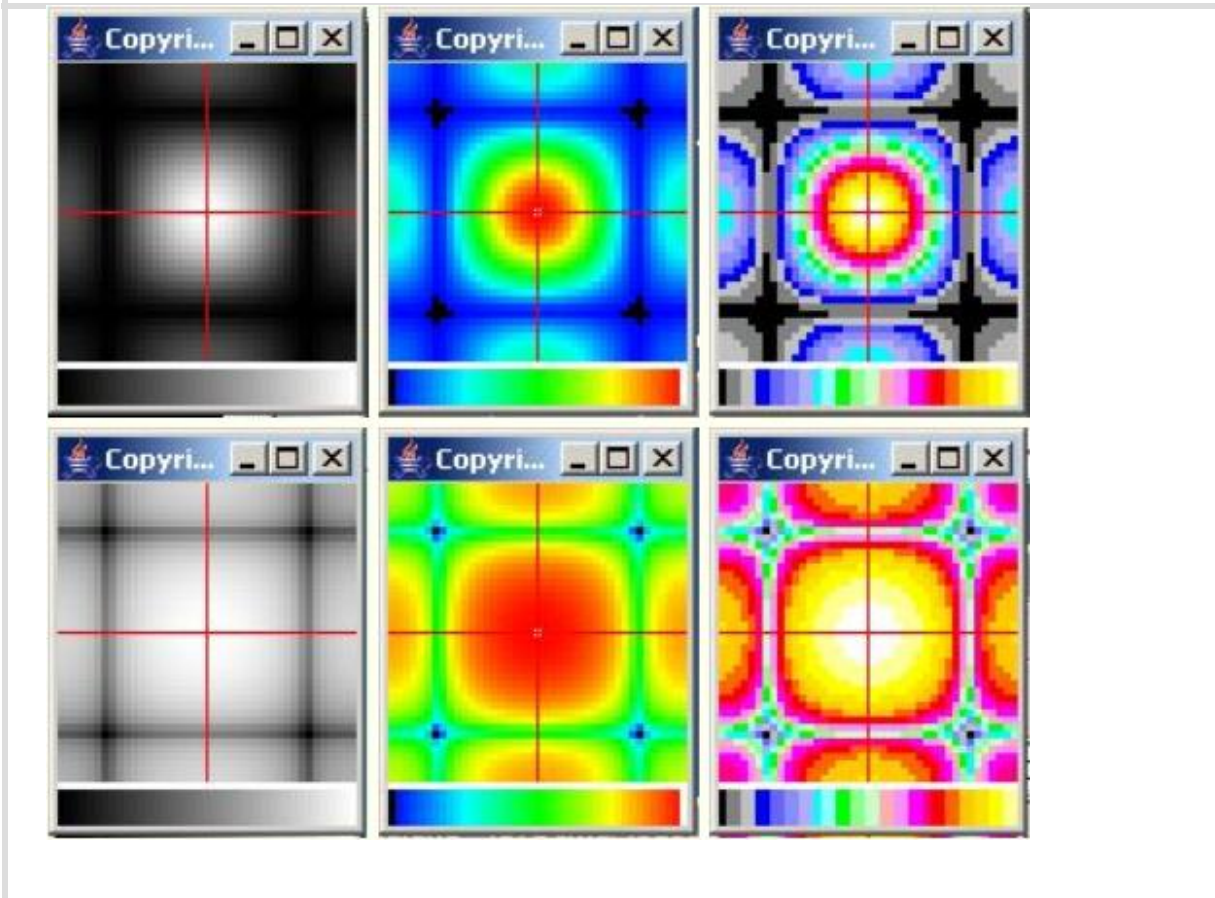
More sample output

The three images in the top row of [Figure 2](#) are reproductions of the three images in [Figure 1](#). They were included in [Figure 2](#) for comparison with the bottom three images in [Figure 2](#).

Figure 2 . Sample output with logarithmic flattening.



Figure 2 . Sample output with logarithmic flattening.



The bottom three images in [Figure 2](#) were produced in exactly the same way as the top three images except that prior to creating the image the elevation values for the surface were converted to the log base 10 of the raw elevation values.

Six different formats

Thus, [Figure 2](#) shows the same 3D surface plotted using six different plotting formats. Going from left to right and top to bottom, the six images illustrate:

- Grayscale (linear)
- Color Shift (linear)
- Color Contour (linear)
- Grayscale with logarithmic data flattening

- Color Shift with logarithmic data flattening
- Color Contour with logarithmic data flattening

Isolates the location of the minima

The significance of the logarithmic conversion can be seen by comparing the two images on the right side of [Figure 2](#). When the raw elevation values were quantized into 23 levels, quite a few of the elevation values were quantized into the minimum value as indicated by the black areas in the top image.

However, after converting the elevation values to logarithmic values, only four points quantized to the minimum value as indicated by the four small black squares in the bottom right image. Thus, the top image on the right shows the general area of the lowest elevations on the surface whereas the bottom image on the right clearly identifies the exact location of each of the four lowest elevations.

A similar discussion holds regarding the two middle plots in [Figure 2](#). The log version on the bottom identifies the location of the minima more closely than does the linear plot at the top.

The log of the surface is flatter

You can also see from the bottom right image of [Figure 2](#) that the central peak of the log of the surface is much broader than the central peak for the raw surface at the top right. This is indicated by the width of the white and yellow areas in the log version as compared to the white and yellow areas in the raw version.

In addition, the elevations of the log data for the minor peaks at the four sides are almost as high as the elevation of the central peak, as indicated by the orange or yellow color at the top of the minor peaks.

The elevations of the tops of the four minor peaks at the corners are perhaps seventy-five percent of the elevation of the central peak as indicated by the pinkish color of those minor peaks in the log version.

(Recall that the elevation of the minor peaks at the corners is only about two levels up from the lowest elevation in the raw surface data.)

All of this flattening was caused by converting the raw surface elevations to the log of the surface elevations before producing the surface plot.

Multiple plotting formats are useful

A plotting format that works best for one surface doesn't necessarily work best for all surfaces. Therefore, it is useful for the program to be able to plot the same surface using different plotting formats.

Extremely easy to use

One of the main objectives in the development of this class was to make it very easy to use. No fancy programming is required to use the class and produce the plots. All that is required is to instantiate an object of the class named **ImgMod29**, passing an array of data to be plotted along with a few other parameters to the constructor. Basically the parameters (*in addition to the data array*) specify:

- Which of the three main formats to use, Grayscale, Color Shift, or Color Contour.
- Whether or not to convert to logarithmic values before plotting.
- Whether or not to draw the red axes shown in [Figure 1](#) and [Figure 2](#).
- How many pixels in the final output should be used to represent a single point on the surface. (*For example, the plots in [Figure 1](#) and [Figure 2](#) use a square of nine pixels to represent each point on the 3D surface.*)

When an object of the class **ImgMod29** is instantiated, everything else happens automatically and the plot is displayed on the computer screen as illustrated by any one of the images in [Figure 1](#) or [Figure 2](#).

Multiple plots

If multiple plots in different formats are needed for a given set of data, all that is required is to instantiate multiple objects of the class named **ImgMod29** passing the same data array with different parameters to the constructor for each plot. Be aware, however, that all of the plots will be produced in a stack in the upper left corner of the screen. You must physically move the plots on the top in order to be able to view the plots lower down in the stack.

Preview

The purpose of the program that I will present and explain in this module is to display a 3D surface using color (or shades of gray) to represent the elevation of each point on a 3D surface.

Constructor

The constructor for this class receives a 3D surface defined as a rectangular 2D array of values of type **double** . Each **double** value represents a sample point on the 3D surface. The surface values may be positive, negative, or both.

When an object of the class is constructed, it plots the 3D surface using one of six possible formats representing the elevation of each point on the surface with a color or a shade of gray.

Parameters

The constructor requires four parameters:

- `double[][] dataIn`
- `int blockSize`
- `boolean axis`
- `int display`

The purpose of each parameter is as follows:

dataIn

The parameter named **dataIn** is a reference to the 2D array of type **double** containing the data that describes the 3D surface.

blockSize

The value of the parameter named **blockSize** defines the size of a colored square in the final display that represents a single input surface elevation value. For example, if **blockSize** is 1, each input surface value is represented by a single pixel in the display. If **blockSize** is 5, each input surface value is represented by a colored square having 5 pixels on each side (*25 pixels in a square*) .

The test code in the **main** method displays a surface having 59 values along the horizontal axis and 59 values along the vertical axis. Each elevation value on the surface is represented in the final display by a colored square that is 2 pixels on each side.

axis

The parameter named **axis** specifies whether optional red axes will be drawn on the display with the origin at the center as shown in [Figure 1](#).

display

The parameter named **display** specifies one of six possible display formats. The value of **display** must be between 0 and 5 inclusive.

Values of 0, 1, and 2 specify the following formats:

display = 0

This value for the **display** parameter specifies a Grayscale plot with a smooth gradient from black at the minimum to white at the maximum.

`display = 1`

This value for the `display` parameter specifies a Color Shift plot with a smooth gradient from blue at the low end through aqua, green, and yellow to red at the high end. The minimum elevation is colored black. The maximum elevation is colored white.

`display = 2`

This value for the `display` parameter specifies a Color Contour plot. The surface is subdivided into 23 levels and each of the 23 levels is represented by one of the following colors in order from lowest to highest elevation:

- `Color.BLACK`
- `Color.GRAY`
- `Color.LIGHT_GRAY`
- `Color.BLUE`
- `new Color(100,100,255)`
- `new Color(140,140,255)`
- `new Color(175,175,255)`
- `Color.CYAN`
- `new Color(140,255,255)`
- `Color.GREEN`
- `new Color(140,255,140)`
- `new Color(200,255,200)`
- `Color.PINK`
- `new Color(255,140,255)`
- `Color.MAGENTA`
- `new Color(255,0,140)`
- `Color.RED`
- `new Color(255,100,0)`
- `Color.ORANGE`
- `new Color(255,225,0)`
- `Color.YELLOW`
- `new Color(255,255,150)`
- `Color.WHITE`

Note that some of the colors in the above list refer to named color constants in the **Color** class. Others refer to new **Color** objects constructed by mixing the specified levels of red, green, and blue.

display = 3, 4, and 5

These values for the display parameter specify that the surface is to be plotted in the same format as for display values 1, 2, and 3, except that the surface elevation values are rectified (*made positive*) and converted to log base 10 before being represented by a color and plotted.

A calibration scale

When the surface is plotted, a horizontal calibration scale is plotted immediately below the surface plot showing the colors used in the surface plot. The colors begin with the color for the lowest elevation at the left and progress to the color for the highest elevation at the right.

Normalization

Regardless of whether the surface elevation values are converted to log values or not, the surface values are normalized to cause them to extend from 0 to 255 before converting the elevation values to color and plotting them. The lowest elevation ends up with a value of 0. The highest elevation ends up with a value of 255.

For display value of 0 or 3

This is a Grayscale plot or a log Grayscale plot as shown at the left side of [Figure 2](#). The highest normalized elevation with a value of 255 is painted white. The lowest normalized elevation with a value of 0 is painted black. The surface is represented using shades of gray.

The shade changes from black to white in a uniform gradient as the normalized surface elevation values progress from 0 to 255.

For display value of 1 or 4

This is a Color Shift plot or log Color Shift plot as shown in the center of [Figure 2](#). The lowest normalized elevation is painted black and the highest normalized elevation is painted white. (*Black and white overwrite blue and red for these two elevation values.*)

The color changes from blue through aqua, green, and yellow to red in a smooth gradient as the normalized surface values progress from 1 to 254. (Values of 0 and 255 would be pure blue and pure red if they were not painted black and white.)

For a display value of 2 or 5

This is a Color Contour plot or log Color Contour plot as shown at the right side of [Figure 2](#). The highest normalized elevation with a value of 255 is painted white. The lowest normalized elevation with a value of 0 is painted black.

The surface is represented using a combination of unique shades of gray and unique colors as the normalized surface elevation values progress from 0 to 255. This is not a gradient display. Rather, the colors in this display change abruptly from one color to the next.

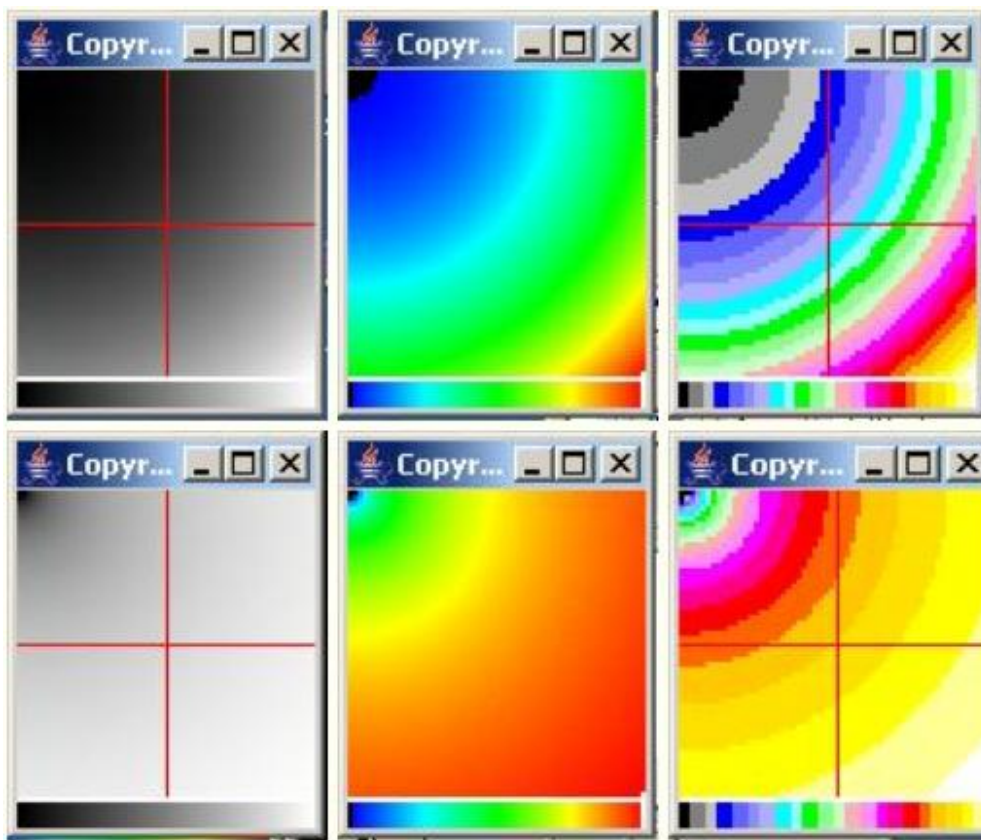
This display format is similar to a contour map where each distinct color traces out a constant elevation level on the normalized surface being plotted.

The main method

Although the class is intended to be used by other programs to display surfaces produced by those programs, the class has a **main** method making it possible to run it in a stand-alone mode for testing.

When the class is run as a stand-alone program, it produces and displays six individual surfaces with the lowest point in the upper left corner and the highest point in the lower right corner. The six images produced by executing the **main** method are shown in [Figure 3](#).

Figure 3 . Program output in self-test mode.



(Remember, all six output images are displayed on top of one another in the upper-left corner of the screen. You must manually move them to see all six images.)

A 3D parabola

The **main** method creates and displays a surface consisting of a 3D parabola. You can think of the surface as representing a one-quarter section of a bowl, or perhaps a satellite dish with the center of the dish in the upper left corner of the image. The top three images in [Figure 3](#) are the images produced from the raw surface. The bottom three images are the images produced by the log of the surface, (*which is no longer a 3D parabola*) .

The calibration scale

The calibration scale is displayed immediately below the image of each surface.

The images are stacked

When the program is executed, the six surfaces are stacked in the upper left corner of the screen. (*You must physically move the images on the top to see the images on the bottom.*) The stacking order of the surfaces from bottom to top is based on the values of the display parameter in the order 0, 1, 2, 3, 4, 5.

With and without axes

Some of the surfaces show axes and some do not. This is controlled by the value of the constructor parameter named **axis** . A true value for **axis** causes the axes to be drawn.

A window listener

The constructor defines an anonymous inner class **WindowListener** on the **close** button on the **Frame** (*the X in the upper right hand corner of the **Frame***) . Clicking the **close** button will terminate the program that uses an object of this class.

Testing

The program was tested using J2SE 8.0 and Win 7. Because the program uses Java features that were introduced in J2SE 5.0, it may not compile successfully with earlier versions of Java.

The program named **ImgMod29**

The program named **ImgMod29** is rather long, so as usual I will break it down and discuss it in fragments. You can view a complete listing of the program in [Listing 29](#) near the end of the module.

The main method

The program consists of a top-level class named **ImgMod29** , plus several inner classes. The class includes a **main** method that is use for self-testing the class. To put things in context, I will begin my discussion with the **main** method.

The main method begins in [Listing 1](#).

Listing 1. Beginning of the main method.

```
public static void main(String[] args){
    int numberRows = 59;
    int numberCols = 59;
    double[][] data =
        new double[numberRows]
[numberCols];
    int blockSize = 2;
```

Local variables

The array for the surface elevation data

[Listing 1](#) declares a 2D array of type **double** to contain the 3D surface elevation data values. This is a square array consisting of 59 elevation values on each side. *(However, there is no requirement for the surface to be square.)*

The `blockSize` parameter

[Listing 1](#) also defines a value of 2 for the **blockSize** parameter. This variable will be passed to the constructor for the **ImgMod29** class, causing each elevation value to be plotted as a small square of four pixels, two pixels on each side of the square.

*(The overall size of the display can be controlled by controlling the size of the array containing the surface elevation values and also controlling the value of **blockSize** .)*

A 3D parabolic surface

The array of surface elevation data is populated by the code in [Listing 2](#).

Listing 2. A 3D parabolic surface,

Listing 2. A 3D parabolic surface,

```
for(int row = 0;row < numberRows;row++){  
    for(int col = 0;col < numberCols;col++){  
        int xSquare = col * col;  
        int ySquare = row * row;  
        data[row][col] = xSquare + ySquare;  
    }//end col loop  
}//end row loop
```

I will allow you to evaluate this code on your own. It creates a 3D surface with the lowest elevation at the upper left corner and the highest elevation at the lower right corner. The surface is a one-quarter section of a 3D parabola, as shown in [Figure 3](#).

Display six surface images

[Listing 3](#) shows the code that causes the 3D surface elevation data to be displayed as six independent images (*each statement in [Listing 3](#) produces one output image*) .

IMPORTANT: this is the same code that would be used by other programs to incorporate this 3D surface plotting capability into those programs.

Listing 3. Display six surface images.

Listing 3. Display six surface images.

```
new ImgMod29(data, blockSize, true, 0);  
new ImgMod29(data, blockSize, false, 1);  
new ImgMod29(data, blockSize, true, 2);  
new ImgMod29(data, blockSize, true, 3);  
new ImgMod29(data, blockSize, false, 4);  
new ImgMod29(data, blockSize, true, 5);
```

Just instantiate an object

All that is necessary for another program to incorporate this class to display a 3D surface is to instantiate an object of the class named **ImgMod29** passing four parameters to the constructor.

The parameters

The first parameter is a reference to the 2D array of type **double** containing the surface elevation values.

The second parameter is the **blockSize** . This **int** value specifies the size of one side of a square of pixels, (*all of the same color*) that will be used to represent each surface elevation value in the final display. As mentioned earlier, this value was set to 2 in the sample displays produced by the **main** method. However, it could have been any reasonable value such as 5, 10, or 15 for example.

The third parameter is true if you want the red axes to be drawn and is false otherwise (*as shown in [Figure 3](#)*) .

The fourth parameter is an integer value between 0 and 5 inclusive, which specifies the plotting format as follows:

- 0 - Grayscale (linear)

- 1 - Color Shift (linear)
- 2 - Color Contour (linear)
- 3 - Grayscale with logarithmic data conversion before plotting
- 4 - Color Shift with logarithmic data conversion before plotting
- 5 - Color Contour with logarithmic data conversion before plotting

Instantiate six separate objects

[Listing 3](#) instantiates six such objects to display the same 3D surface, one for each plotting format as shown in [Figure 3](#). Some of the objects display the axes and others do not. All use a `blockSize` value of 2.

Each display object appears in the upper-left corner of the screen. Thus, when two or more such objects are instantiated, they appear as a stack. It is necessary to physically move those on top of the stack to see those further down.

[Listing 3](#) signals the end of the main method.

The `ImgMod29` class

[Listing 4](#) shows the beginning of the class named `ImgMod29` and the beginning of the constructor for the class.

Listing 4. Beginning of the `ImgMod29` class.

Listing 4. Beginning of the ImgMod29 class.

```
class ImgMod29 extends Frame{
    int dataWidth;
    int dataHeight;
    int blockSize;
    boolean axis;
    double[][] data;

    ImgMod29(double[][] dataIn,int blockSize,
            boolean axis,int display){
        //Get and save several important values
        this.blockSize = blockSize;
        this.axis = axis;
        dataHeight = dataIn.length;
        dataWidth = dataIn[0].length;
        boolean logPlot = false;
        int displayType = display;
```

The meaning and purpose of each of the constructor parameters was explained earlier, so I won't repeat that explanation here. The code in [Listing 4](#) is straightforward and should not require further explanation.

Establish display format for log conversion

The code in [Listing 5](#) uses the incoming value of the **display** parameter to establish the display format if the value of **display** is 3, 4, or 5.

Listing 5. Establish display format for log conversion.

```
if(display == 3){
    displayType = 0;
    logPlot = true;
}else if(display == 4){
    displayType = 1;
    logPlot = true;
}else if(display == 5){
    displayType = 2;
    logPlot = true;
}else if((display > 5) || (display < 0)){
    System.out.println(
        "DisplayType input error,
terminating");
    System.exit(0);
} //end if
```

The default display format is one of the three basic types **without log conversion** of the surface elevation data. (The value of **logPlot** is set to *false* in [Listing 4](#).) If the incoming parameter value is 3, 4, or 5, the code in [Listing 5](#) establishes the display format as one of the three basic types **with log conversion** of the surface elevation data prior to plotting.

(Note that the code in [Listing 5](#) sets the value of the variable named **logPlot** to true. The value stored in this variable will be used later to determine if log conversion of the elevation data is required.)

The three basic types

The three basic types are:

- displayType = 0, Grayscale plot
- displayType = 1, Color Shift plot
- displayType = 2, Color Contour plot

These three basic types without log data conversion are shown from left to right in [Figure 1](#). The three basic types are shown with log conversion from left to right in the bottom rows of [Figure 2](#) and [Figure 3](#).

Copy the input elevation data

[Listing 6](#) makes a working copy of the input data to avoid damaging the original data. This is done to protect the data belonging to the program that instantiates an object of the class **ImgMod29**.

Listing 6. Copy the input elevation data.

```
data = new double[dataHeight][dataWidth];
for(int row = 0; row < dataHeight; row++){
    for(int col = 0; col < dataWidth; col++){
        data[row][col] = dataIn[row][col];
    }//end loop on col
}//end loop on row
```

Convert to log data if required

The code in [Listing 7](#) uses the **log10** method of the **Math** class to perform a log conversion of the surface elevation data if the value of **logPlot** is true.

Listing 7. Convert to log data if required

```
if(logPlot){//Convert to log base 10.
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            //Change the sign on negative values
            // before converting to log values.
            if(data[row][col] < 0){
                data[row][col] = -data[row][col];
            }//end if
            if(data[row][col] > 0){
                //Convert value to log base 10. Log
                // of 0 is undefined. Just leave it
                // at 0.
                data[row][col] =
                    Math.log10(data[row]
[col]);
            }//end if
        }//end col loop
    }//end row loop
}//end if on logPlot
```

New to J2SE 5.0

According to Oracle's documentation, the **log10** method became part of Java with the release of J2SE 5.0. Therefore, this code is not compatible with earlier versions of Java.

Here are a couple of restrictions taken from Oracle's documentation that apply to the use of the method named **log10** :

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive zero or negative zero, then the result is negative infinity.

Because of the first restriction, the code in [Listing 7](#) converts all negative values into positive values before performing the conversion. Because of the second restriction, no attempt is made to compute the log of elevation values of zero.

Otherwise, the code in [Listing 7](#) is straightforward and should not require further explanation.

Normalize the surface elevation data

After converting the elevation data to log form, *(or not converting as the case may be)*, the code in [Listing 8](#) calls the method named **scaleTheSurfaceData** to normalize the elevation data by squeezing it into the integer range between 0 and 255 inclusive. When this method returns, the lowest elevation has a value of 0 and the highest elevation has a value of 255.

Listing 8. Normalize the surface elevation data.

```
scaleTheSurfaceData( );
```

The elevation data is normalized to this range to make it easier later to form relationships between the elevation values and allowable color values.

(Recall that allowable color values range from 0 to 255.)

The method named scaleTheSurfaceData

At this point, we will take a side trip and examine the method named **scaleTheSurfaceData** , which is shown in its entirety in [Listing 9](#).

Listing 9. The method named scaleTheSurfaceData.

```
double min;
double max;
//This method is used to scale the surface
data
// to force it to fit in the range from 0 to
// 255.
void scaleTheSurfaceData(){
    //Find the minimum surface value.
    min = Double.MAX_VALUE;
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            if(data[row][col] < min)
                min = data[row][col];
        }//end col loop
    }//end row loop

    //Shift all values up or down to force new
    // minimum value to be 0.
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            data[row][col] = data[row][col] - min;
        }//end col loop
    }//end row loop

    //Now get the maximum value of the shifted
    // surface values
    max = -Double.MAX_VALUE;
```

Listing 9. The method named scaleTheSurfaceData.

```
for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
        if(data[row][col] > max)
            max = data[row][col];
    }//end col loop
}//end row loop

//Now scale all values to cause the new
// maximum value to be 255.
for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
        data[row][col] =
            data[row][col] * 255/max;
    }//end col loop
}//end row loop
}//end scaleTheSurfaceData
```

While this method is rather long, it is completely straightforward and shouldn't require further explanation.

Create an appropriate pair of Canvas objects

Return now to the discussion of the constructor for the **ImgMod29** class.

The code in [Listing 10](#) uses the value of **displayType** to make a decision and to instantiate a pair of objects from two of six different inner classes, each of which extends the class named **Canvas** . One of the objects in the pair is used to display the 3D surface according to a specified format. The other object in the pair is used to display the calibration strip below the surface display.

Listing 10. Create an appropriate pair of Canvas objects.

```
Canvas surface = null;
Canvas scale = null;

//Establish the format based on the value of
// the parameter named display.
if(displayType == 0){
    //Create a type 0 Canvas object to draw
the
    // surface on. This is a Grayscale plot
    // display.
    surface = new CanvasType0surface();
    //Create a Canvas object to draw the scale
    // on for the Grayscale plot.
    scale = new CanvasType0scale();
}else if(displayType == 1){
    //Color Shift plot
    surface = new CanvasType1surface();
    scale = new CanvasType1scale();
}else if(displayType == 2){
    //Color Contour plot.
    surface = new CanvasType2surface();
    scale = new CanvasType2scale();
} //end if-else on display type
```

The code in [Listing 10](#) is straightforward and shouldn't require further explanation.

The interesting code

The interesting code is contained in the overridden **paint** method belonging to each of the six inner classes from which the pair of objects is instantiated. I will explain those overridden paint methods later.

Add the Canvas objects to the Frame

The default layout manager for a **Frame** object is **BorderLayout** . The code in [Listing 11](#) adds one of the above-instantiated objects to the center location of the **Frame** , and adds the other object to the **South** location of the **Frame** . This produces the format with the surface plot above the calibration scale as shown by any of the images in [Figure 1](#) through [Figure 3](#).

Listing 11. Add the Canvas objects to the Frame.

```
//Add the plotted surface to center of the
// Frame
add(BorderLayout.CENTER,surface);
//Add the scale to bottom of Frame
add(BorderLayout.SOUTH,scale);
//Cause the size of the Frame to be just
// right to contain the two Canvas objects.
pack();

//Set Frame cosmetics and make it visible.
setTitle("Copyright 2005 R.G.Baldwin");
setVisible(true);
```

Call the pack method and set the title

After adding the two objects to the **Frame** , [Listing 11](#) calls the **pack** method on the **Frame** to cause the size of the **Frame** to close in around the two objects.

Finally, [Listing 11](#) sets the title on the **Frame** and makes the s visible.

Register an anonymous WindowListener object

[Listing 12](#) registers an anonymous **WindowListener** object on the **Frame** to cause the program to terminate whenever the user clicks the X-button in the upper right corner of the **Frame** .

Listing 12. Register an anonymous WindowListener object.

```
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        } //end windowClosing
    } //end class definition
} //end addWindowListener

} //end constructor
```

If you are unfamiliar with the use of anonymous inner classes, you can learn about such topics in my other publications.

[Listing 12](#) also signals the end of the constructor for the class named **ImgMod29** .

Six different inner classes

We have finally gotten to the fun part of the program. This is the part where we write the code that determines how the surface elevations and the calibration scale values are displayed. This part of the program consists of six different inner classes.

The fact that the classes are inner classes makes it possible for methods in the class to access instance variables and methods of the containing object of type **ImgMod29** . This makes the programming somewhat easier than would be the case if they were all top-level classes.

Subclasses of the Canvas class

Each of the six inner classes is a subclass of the class named **Canvas** and each of the classes overrides the **paint** method. The code in the overridden **paint** methods for the classes that display the 3D surfaces

- access the surface elevation data,
- convert that data into colors and,
- display those colors in the correct locations on the screen.

The names and behaviors of three of those three classes are:

- **CanvasType0surface** - Displays a Grayscale plot
- **CanvasType1surface** - Displays a Color Shift plot
- **CanvasType2surface** - Displays a Color Contour plot

Associated directly with the three above classes are three other inner classes that are used to display the calibration scale immediately below the plot of the 3D surface. The names and behaviors of those three classes are:

- **CanvasType0scale** - Displays a Grayscale calibration scale
- **CanvasType1scale** - Displays a Color Shift calibration scale
- **CanvasType2scale** - Displays a Color Contour calibration scale

The **getCenter** method

Before getting into the details of these inner classes, however, I will present and briefly discuss a method named **getCenter** , which is called by the constructor for each of the surface plotting classes.

The **getCenter** method is used to find the horizontal and vertical center of the surface. These values are used to position the optional red axes that may be

drawn on the surface. This method is shown in its entirety in [Listing 13](#).

Listing 13. The method named getCenter.

```
int horizCenter;
int vertCenter;

void getCenter(){
    if(dataWidth%2 == 0){//even
        horizCenter =
            dataWidth * blockSize/2 +
blockSize/2;
    }else{//odd
        horizCenter = dataWidth * blockSize/2;
    }//end else

    if(dataHeight%2 == 0){//even
        vertCenter =
            dataHeight * blockSize/2 +
blockSize/2;
    }else{//odd
        vertCenter = dataHeight * blockSize/2;
    }//end else
} //end getCenter
```

Even or odd is very important

Note that the returned values depend on whether the dimensions of the surface are even or odd.

(For example, the center of a string of five blocks of pixels is the third block whereas the center of a string of six blocks of pixels is half way between the third and fourth blocks.)

Now that you know about the difference between even and odd surface dimensions, the code in [Listing 13](#) should be straightforward and should not require further discussion.

Grayscale plot format

[Listing 14](#) shows the beginning of the class named **CanvasType0surface** and also shows the entire constructor for that class.

An object of this class is used to paint a **Grayscale** plot of a 3D surface ranging from black at the lowest elevation to white at the highest elevation. The various shades of gray vary in a smooth gradient between the two extremes. The leftmost image in [Figure 1](#) is an example of the type of plot produced by an object of this class.

Listing 14. Beginning of the class named CanvasType0surface.

```
class CanvasType0surface extends Canvas{
    CanvasType0surface(){//constructor
        setSize(dataWidth * blockSize,
                dataHeight * blockSize);
        getCenter();
    }//end constructor
```


The constructor

The constructor for the class is straightforward. It accesses instance variables of the outer enclosing object to set the size of the canvas on the basis of the size of the surface and the size of the square of pixels that will be used to represent each elevation value on the surface.

The constructor also calls the **getCenter** method to get the coordinates of the center of the surface in order to be able to draw the optional red axes in the correct position later.

Overridden paint method for CanvasType0surface class

The real work is done by the overridden **paint** method, which begins in [Listing 15](#). Of the three classes used to plot the 3D surface, this is the simplest.

Listing 15. Beginning of overridden paint method.

```
public void paint(Graphics g){
    Color color = null;
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            //Add in red, green, and blue in
            // proportion to the value of the
            // surface height.
            int red = (int)data[row][col];
            int green = red;
            int blue = red;
```

Purpose of overridden paint method

The purpose of this overridden **paint** method is to convert the elevation value of each point on the 3D surface into an appropriate shade of gray, and to paint a square of pixels on the screen at that elevation value's location where every pixel in the square is the same shade of gray.

In order to produce a gray pixel in a 24-bit color system, you need to set the color values for red, green, and blue to the same value. If all three color values are zero, the pixel color is black. If all three color values are 255, the pixel color is white. If all three color values fall somewhere in between zero and 255, the pixel color will be some shade of gray.

Recall that the elevation values were earlier normalized to fall in the range from zero to 255. The code in [Listing 15](#) uses a nested **for** loop to access every elevation value in the array that describes the 3D surface. Then it sets the red, green, and blue color values to the normalized surface value for each point on the 3D surface.

Instantiate a Color object

Continuing inside the nested **for** loops, the code in [Listing 16](#) instantiates a new object of type **Color** based on the current red, green, and blue color values.

Listing 16. Instantiate a Color object.

```
color = new Color(red,green,blue);
```

Set colors and draw squares

The code in [Listing 17](#) calls the **setColor** method of the **Graphics** class to set the current drawing color to the color described by the **Color** object referred to by the reference variable named **color**.

Listing 17. Set colors and draw squares.

```
//Set the color value.
g.setColor(color);
//Draw a square of the specified size
//in the specified color at the
// specified location.
g.fillRect(col * blockSize,
           row * blockSize,
           blockSize,
           blockSize);
} //end col loop
} //end row loop
```

Paint a colored square

Finally the code in [Listing 17](#) calls the **fillRect** method of the **Graphics** class to paint a square of pixels of the specified size at the specified location in the specified color.

This process is repeated for every elevation point on the 3D surface data, producing an output similar to the leftmost image in [Figure 1](#).

[Listing 17](#) signals the end of the nested **for** loops. When the code in [Listing 17](#) finishes execution, the 3D surface has been plotted, but it does not yet

contain the optional red axes.

Draw the optional red axes

[Listing 18](#) tests to see if the value of the axis parameter is true. If so, it uses the information obtained earlier from the **getCenter** method, along with the **setColor** and **drawLine** methods of the **Graphics** class to draw the optional red axes shown in the images in [Figure 1](#). These axes always intersect at the center of the image.

Listing 18. Draw the optional red axes.

```
        if(axis){
            g.setColor(Color.RED);
            g.drawLine(0,vertCenter,2*horizCenter,
vertCenter);
            g.drawLine(horizCenter,0,horizCenter,
2*vertCenter);
        }//end if
    }//end paint
} //end inner class CanvasType0surface
```

[Listing 18](#) also signals the end of the overridden **paint** method and the end of the inner class named **CanvasType0Surface**.

Beginning of the class named **CanvasType0scale**

An object of the class named **CanvasType0scale** is used to plot the calibration scale that is displayed immediately below the surface plot for the **Grayscale** plot format. The beginning of this class and the constructor for this class are shown in [Listing 19](#).

Listing 19. Beginning of the class named CanvasType0scale .

```
class CanvasType0scale extends Canvas{
    //Set the physical height of the scale strip
    // in pixels.
    int scaleHeight = 6 * blockSize;

    CanvasType0scale(){//constructor
        //Set the size of the Canvas based on the
        // width of the surface and the size of
the
        // square used to represent each value on
        // the surface.
        setSize(dataWidth *
blockSize, scaleHeight);
    }//end constructor
```

How it works

Basically this class (*as well as the other two classes that create calibration scales*) operates by constructing an artificial surface, (*which is like a long thin board*) , positioned such that one end has an elevation of 0 and the other end has an elevation of 255. The length of this long thin surface is equal to the width of the surface plot for the Grayscale plot format.

The same Grayscale color algorithm is applied to this artificial surface that is applied to the real surface. The result is a linear representation of the colors produced by the color algorithm from the lowest elevation at 0 to the highest elevation at 255. This result is displayed immediately below the real surface with the lowest elevation at the left end and the highest elevation at the right end. An example is shown in the leftmost image in [Figure 1](#).

The code in [Listing 19](#) establishes the size of the calibration scale surface.

The overridden paint method

[Listing 20](#) shows the overridden **paint** method that is used to plot the calibration scale for the Grayscale plot format.

Listing 20. The overridden paint method.

```
public void paint(Graphics g){
    //Vary from white to black going from 255
    // to 0.
    Color color = null;
    //Don't draw in top row. Leave it blank to
    // separate the scale strip from the
    // drawing of the surface above it.
    for(int row = 1; row < scaleHeight; row++){
        for(int col = 0; col < dataWidth; col++){

            //Compute the value of the scale
            // surface.
            int scaleValue = 255 * col /
                                (dataWidth -
1);
```

Listing 20. The overridden paint method.

```
        //See the class named
        // CanvasType0surface for explanatory
        // comments regarding the following
        // color algorithm.
        int red = scaleValue;
        int green = red;
        int blue = red;
        color = new Color(red,green,blue);
        g.setColor(color);
        g.fillRect(col * blockSize,
                    row * blockSize,
                    blockSize,
                    blockSize);
    } //end col loop
} //end row loop
} //end paint

} //end inner class CanvasType0scale
```

Because you already understand the color algorithm for the Grayscale plot format, the code in [Listing 20](#) should not require further explanation. This code establishes the elevation level for each point on the calibration surface and paints the box that represents that elevation in the appropriate color.

Color Shift plot format

The **CanvasType1surface** class is used to instantiate an object that represents a normalized 3D surface with the colors ranging from blue at the low elevations through aqua, green, and yellow to red at the high elevations with a smooth gradient from 1 to 254.

(The lowest elevation with a value of 0 is colored black. The highest elevation with a value of 255 is colored white.)

The center image in [Figure 1](#) is an example of this plotting format.

Beginning of the class named CanvasType1surface

The beginning of the class and the constructor for the class named **CanvasType1surface** are shown in [Listing 21](#).

Listing 21. Beginning of the class named CanvasType1surface.

```
class CanvasType1surface extends Canvas{

    CanvasType1surface(){//constructor
        //Set the size of the Canvas based on the
        // size of the surface and the size of the
        // square used to represent each value on
        // the surface.
        setSize(dataWidth * blockSize,
                dataHeight * blockSize);
        getCenter();
    }//end constructor
```

The beginning of each of the three classes that produce the three plotting formats is essentially the same. Therefore, the code in [Listing 21](#) is essentially the same as the code in [Listing 14](#) and should not require further explanation.

The significant differences between the three classes lie in their overridden **paint** methods.

Overridden paint method

The overridden **paint** method for this class, which begins in [Listing 22](#), is probably the most complex of the three.

Listing 22. Beginning of the overridden paint method.

```
public void paint(Graphics g){
    Color color = null;
    for(int row = 0; row < dataHeight; row++){
        for(int col = 0; col < dataWidth; col++){
            int red = 0;
            int green = 0;
            int blue = 0;
```

The **paint** method for this class begins by setting up a pair of nested **for** loops that will be used to process each elevation point on the surface, and by initializing the color values for red, green, and blue to 0 in the innermost loop.

Set white and black for max and min values

If the elevation value is equal to 255, color values are set to cause that elevation to be painted white. If the elevation value is equal to 0, color values are set to cause that elevation to be painted black.

[Listing 23](#) sets the color values to cause the extreme values of 0 and 255 to be painted black and white. Note that the code in [Listing 23](#) is the beginning of a series of **if-else** constructs.

Listing 23. Set white and black for max and min values.

```
if((int)data[row][col] == 255){  
    red = green = blue = 255;//white  
}else if((int)data[row][col] == 0 ){  
    red = green = blue = 0;//black
```

Elevations other than the extreme ends

If the elevation is not one of the extreme values of 0 or 255, control passes to code that subdivides the total elevation range from 1 to 254 into the following four ranges and then sets the color values for each range separately:

- 0 less than elevation less than or equal 63
- 63 less than elevation less than or equal 127
- 127 less than elevation less than or equal 191
- 191 less than elevation less than or equal 254

Processing the lowest range

[Listing 24](#) shows the code that is used to process the lowest range of elevations between 1 and 63.

Listing 24. Process elevations from 1 to 63 inclusive.

```
        }else if(((int)data[row][col] > 0) &&
                ((int)data[row][col] <= 63))
    {
        int temp = 4 * ((int)data[row][col]
                        -
0);
        blue = 255;
        green = temp;
```

What we are shooting for here is to produce color values that will result in a smooth gradient of color from blue at the low end to aqua at the high end of the range.

(See the leftmost one-fourth of the calibration scale for the middle image in [Figure 1](#).)

Scale the elevation values

[Listing 24](#) begins by multiplying the elevation value by a factor of 4 to put it into the range from 4 to 252. This makes the elevation values compatible with allowable color values that range from 0 to 255.

The color aqua

The color aqua is produced by mixing equal amounts of blue and green. [Listing 24](#) holds the value of blue constant at 255 and increases the value of green in proportion to the elevation value. Thus, at the lower end of the range, blue has a value of 255 and green has a value of 4. *(This is almost pure blue.)*

At the upper end of the range, blue still has a value of 255 and green has a value of 252. (*This is almost the pure secondary color aqua.*)

In all cases, the value of red is 0 within this range. These color values (*blue and temp*) will be used later to instantiate a **Color** object, which will be used to control the plotting color for that portion of the display.

Processing the other three ranges

Now that you know the basic scheme, you shouldn't have any difficulty understanding the code for processing the other three ranges shown in [Listing 25](#).

Listing 25. Processing the other three ranges.

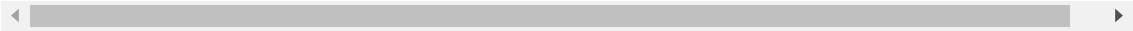
Listing 25. Processing the other three ranges.

```
        }else if(((int)data[row][col] > 63) &&
                ((int)data[row][col] <= 127))
{
    int temp = 4 * ((int)data[row][col]
-
64);
    green = 255;
    blue = 255 - temp;

    }else if(((int)data[row][col] > 127)
&&
                ((int)data[row][col] <= 191))
{
    int temp = 4 * ((int)data[row][col]
-
128);
    green = 255;
    red = temp;

    }else if(((int)data[row][col] > 191)
&&
                ((int)data[row][col] <= 254))
{
    int temp = 4 * ((int)data[row][col]
-
192);
    red = 255;
    green = 255 - temp;

    }//end else
```



Gradient from aqua to green

The second range produces a smooth gradient from aqua to green. In this range, the green color value is held constant at 255 and the blue color value is caused to decrease in inverse proportion to the normalized color value.

Gradient from green to yellow

The third range produces a smooth gradient from green to yellow. (*Yellow is produced by mixing equal amounts of red and green.*) Within this range, green is held constant at a value of 255 and the value of red increases in direct proportion to the normalized elevation value.

Gradient from yellow to red

The fourth range produces a smooth gradient from yellow to red. Within this range, the value of red is held constant at 255 and the value of green decreases in inverse proportion to the normalized elevation value.

A homework assignment

A useful homework assignment would be for you to modify the program as follows:

Subdivide the total range into eight sub ranges instead of four as I did. Choose four additional colors that you can produce by mixing various levels of red, green, and blue. Modify the code to cause the colors to vary with a smooth gradient through those eight colors in succession.

The rest of the overridden paint method

The rest of the overridden **paint** method for this class is essentially the same as the code that I explained in [Listing 16](#) through [Listing 18](#) . Having set the current plotting color, the method goes on to paint a square of pixels in that color at the correct location. Then it draws the optional red axes if specified. Therefore, I won't repeat that explanation. You can view this code in [Listing 29](#) near the end of the module.

The class named CanvasType1scale

The inner class named **CanvasType1scale** is used to construct a color scale that matches the color algorithm used in the class named **CanvasType1surface** .

The overridden **paint** method for this class replicates the color algorithm in the overridden **paint** method for the **CanvasType1surface** class.

Except for the difference in the overridden **paint** method, the structure of this class is the same as the class named **CanvasType0scale** , which I discussed earlier beginning with [Listing 19](#) . Therefore, I won't repeat that discussion here. You can view the class in [Listing 29](#) near the end of the module.

Color Contour plot format

The class named **CanvasType2surface** is an inner class used to instantiate an object that plots a surface where each elevation on the surface is represented by a color taken from a color palette containing a finite number of colors. As written, the color palette contains 23 different colors and shades of gray, but you can easily increase that number if you would like to do so.

The color palette

Before getting into the details of the class, I will explain the color palette. The color palette is produced by a method named **getColorPalette** , shown in its

entirety in [Listing 26](#). This is a utility method that is called by the inner classes named **CanvasType2surface** and **CanvasType2scale** .

Listing 26. The method named getColorPalette.

Listing 26. The method named getColorPalette.

```
Color[] getColorPalette(){
    //Note that the following is an initialized
    // 1D array of type Color.
    Color[] colorPalette = {
        Color.BLACK,//          0,   0,   0
        Color.GRAY,//          128,128,128
        Color.LIGHT_GRAY,//    192,192,192
        Color.BLUE,//           0,   0,255
        new Color(100,100,255),//100,100,255
        new Color(140,140,255),//140,140,255
        new Color(175,175,255),//175,175,255
        Color.CYAN,//           0,255,255
        new Color(140,255,255),//140,255,255
        Color.GREEN,//          0,255,   0
        new Color(140,255,140),//140,255,140
        new Color(200,255,200),//200,255,200
        Color.PINK,//           255,175,175
        new Color(255,140,255),//255,140,255
        Color.MAGENTA,//        255,   0,255
        new Color(255,0,140),   //255,   0,140
        Color.RED,//            255,   0,   0
        new Color(255,100,0),//   255,100,   0
        Color.ORANGE,//         255,200,   0
        new Color(255,225,0),//   255,225,   0
        Color.YELLOW,//         255,255,   0
        new Color(255,255,150),//255,255,150
        Color.WHITE};//         255,255,255

    return colorPalette;
} //end getColorPalette
```

The purpose of this method is to establish a color palette containing references to **Color** objects representing 23 distinct colors and shades of gray. The references are stored in a one-dimensional array object as element type **Color**. The values shown in comments in [Listing 26](#) represent the values of red, green, and blue required to produce that specific color.

As you can see, some of the elements in the array refer to **Color** objects defined as named constants (*public final variables*) in the **Color** class. Other elements in the array refer to **Color** objects that are instantiated using red, green, and blue color values of my own choosing. The actual colors represented by these objects, going from top to bottom, match the colors shown in the calibration scale for the rightmost image in [Figure 1](#).

Rearrange and add new colors

If you would like to do so, you can rearrange the colors in the array. This will result in different colors being adjacent to one another in the calibration scale. Also if you would like to do so, you can remove colors from the array or add new colors of your own choosing to the array. The overridden paint methods in the classes named **CanvasType2surface** and **CanvasType2scale** are designed to take such changes into account.

The CanvasType2surface class

You can view the entire class named **CanvasType2surface** in [Listing 29](#) near the end of the module. Because of the similarity of this class to others that I have previously discussed, I will limit my discussion to the portions of the overridden **paint** method that distinguish this class from the others.

As it turns out, this is perhaps the simplest of the three overridden **paint** methods. The method begins in [Listing 27](#) where the **getColorPalette** method is called to get a reference to the color palette discussed above.

Then a pair of nested **for** loops is set up to process every elevation value on the 3D surface.

Listing 27. Beginning of overridden paint method.

```
public void paint(Graphics g){
    Color[] colorPalette = getColorPalette();

    for(int row = 0; row < dataHeight; row++){
        for(int col = 0; col < dataWidth; col++){

            int quantizedData = (int)(Math.round(
                data[row][col]*(
                    colorPalette.length-
1)/255));
```

Quantize the elevation levels

The code in [Listing 27](#) quantizes the elevation levels into a set of integer values ranging from 0 to one less than the number of elements in the color palette. As written, this redefines the normalized elevation values as extending from 0 to 22, instead of from 0 to 255.

(If you change the length of the color palette, the number of ranges will change accordingly.)

Set the color value

The code in [Listing 28](#) uses the quantized elevation value to index into the color palette and retrieve a reference to a **Color** object. This reference is passed to the **setColor** method setting the current plotting color to the color represented by that index value.

Listing 28. Set the color value.

```
g.setColor(colorPalette[quantizedData]);
```

Having set the current plotting color, as in the other two cases discussed earlier, the method goes on to paint a square of pixels in that color at the correct location. Then it draws the optional axes if specified. The code to accomplish these operations is the same as code discussed previously, so I won't repeat that discussion here. You can view the code in [Listing 29](#) near the end of the module.

The class named `CanvasType2scale`

This inner class is used to construct a color scale that matches the color algorithm used in the class named `CanvasType2surface`. Except for the difference in the overridden **paint** method, this class is essentially the same as the other two classes used to construct color scale objects. Therefore, I won't repeat that discussion.

You can view the class in its entirety in [Listing 29](#) near the end of the module. You can view the graphic output produced by this class in the calibration scale for the image at the rightmost end of [Figure 1](#).

Run the program

I encourage you to copy, compile, and run the program that you will find in [Listing 29](#). Modify the program and experiment with it in order to learn as much as you can about the use of Java for displaying 3D data.

A better color scheme

See if you can come up with a better color scheme than the color schemes that I used in my version of the program. For example, you might add new colors

to the color palette used for the Color Contour plot. That will be very easy to do. All you need to do is add them to the array.

(The hard part will be to identify new colors that are visually separable from the colors that are already being used.)

You might also add new colors to the color algorithm for the Color Shift plot. This will be somewhat more difficult in that additional coding will be required to incorporate those new colors.

Create different test surfaces

You might also want to modify the code in the **main** method to cause it to create different test surfaces. You could even write new independent programs that create surfaces and use this class named **ImgMod29** to plot those surfaces. Remember, all that's necessary to use this class to plot your own 3D surface is to include a statement similar to the following in your code:

```
new ImgMod29(data, blockSize, true, 0);
```

Create larger test surfaces with a smaller blockSize

It was necessary for me to keep the images in this module small in order to force them to fit into this publication format. As you are experimenting, make your test surfaces larger and your **blockSize** smaller. This will result in smoother edges where different colors meet.

The parameters to the ImgMod29 constructor

The parameter named **data** in the above example is a reference to a 2D array of type **double** that describes the surface to be plotted.

The second parameter named **blockSize** specifies the size of one side of the square of pixels in the final plot that you want to use to represent each elevation point on your 3D surface. Set this to 0 if you are unsure as to what size square you need.

The third parameter specifies whether or not you want to have the optional red axes drawn. A value of true causes the axes to be drawn. A value of false causes the axes to be omitted.

The fourth parameter is an integer that specifies the plotting format as follows:

- 0 - Grayscale (linear)
- 1 - Color Shift (linear)
- 2 - Color Contour (linear)
- 3 - Grayscale with logarithmic data conversion
- 4 - Color Shift with logarithmic data conversion
- 5 - Color Contour with logarithmic data conversion

The class couldn't be simpler to use.

Above all, have fun and learn as much as you can in the process.

Summary

In this module, I explained and illustrated a program for using Java and color to plot 3D surfaces. The program is extremely easy to use and makes it easy to plot your surface using six different plotting formats in a wide range of sizes.

Complete program listing

A complete listing of the program is provided in [Listing 29](#) below.

Listing 29. Source code for ImgMod29.java.

```
/*File ImgMod29.java  
Copyright 2005, R.G.Baldwin
```

Listing 29. Source code for ImgMod29.java.

The purpose of this program is to display a 3D surface using color to represent the height of each point on the surface.

The constructor for this class receives a 3D surface defined as a rectangular 2D array of double values. The surface values may be positive or negative or both. When an object of the class is constructed, it draws the 3D surface using one of six possible formats representing the height of each point on the surface with a color.

The constructor requires four parameters:

```
double[][] dataIn
int blockSize
boolean axis
int display
```

The purpose of each parameter is as follows:

dataIn - The parameter named dataIn is a reference to the array containing the data that describes the 3D surface.

blockSize - The value of the parameter named blockSize defines the size of a colored square in the final display that represents an input surface value. For example, if blockSize is 1, each input surface value will be represented by a single pixel in the display. If blockSize is 5, each input surface value will be represented by a colored square having 5 pixels on each side. For example, the test code in the main method

Listing 29. Source code for ImgMod29.java.

displays a surface having 59 values along the horizontal axis and 59 values along the vertical axis. Each value on the surface is represented in the final display by a colored square that is 2 pixels on each side.

axis - The parameter named axis specifies whether red axes will be drawn on the display with the origin at the center.

display - The parameter named display specifies one of six possible display formats. The value of display must be between 0 and 5 inclusive. Values of 0, 1, and 2 specify the following formats:

0 - Gray scale gradient from black at the minimum to white at the maximum.

1 - Color gradient from blue at the low end through aqua, green, yellow to red at the high end. The minimum value is colored black. The maximum value is colored white..

2.- The surface is subdivided into 23 levels and each of the 23 levels is represented by one of the following Color Contour plot in order

from minimum to maximum.

Color.BLACK

Color.GRAY

Color.LIGHT_GRAY

Color.BLUE

new Color(100,100,255)

new Color(140,140,255)

new Color(175,175,255)

Color.CYAN

new Color(140,255,255)

Listing 29. Source code for ImgMod29.java.

```
Color.GREEN  
new Color(140,255,140)  
new Color(200,255,200)  
Color.PINK  
new Color(255,140,255)  
Color.MAGENTA  
new Color(255,0,140)  
Color.RED  
new Color(255,100,0)  
Color.ORANGE  
new Color(255,225,0)  
Color.YELLOW  
new Color(255,255,150)  
Color.WHITE
```

Values of 3, 4, and 5 for the parameter named display draw the surface in the same formats as above except that the surface values are first rectified and then converted to log base 10 values before being converted to color and drawn.

When the surface is drawn, a horizontal scale strip is drawn immediately below the surface showing the colors used in the drawing starting with the color for the minimum at the left and progressing to the color for the maximum at the right.

Regardless of whether the surface values are converted to log values or not, the surface values are normalized to cause them to extend from 0 to 255 before converting to color and drawing.

For a display value of 0 or 3, the highest point with a value of 255 is painted white. The lowest

Listing 29. Source code for ImgMod29.java.

point with a value of 0 is painted black, The surface is represented using shades of gray. The shade changes from black to white in a uniform gradient as the height of the normalized surface values progress from 0 to 255.

For a display value of 1 or 4, the lowest point is painted black and the highest point is painted white. The color changes from blue through aqua, green, and yellow to red in a smooth gradient as the normalized surface values progress from 1 to 254. (Values of 0 and 255 would be pure blue and pure red if they were not overridden by black and white.)

For a display value of 2 or 5, the highest point with a value of 255 is painted white. The lowest point with a value of 0 is painted black, The surface is represented using a combination of unique shades of gray and unique colors as the normalized surface values progress from 0 to 255. This is not a gradient display. Rather, this display format is similar to a contour map where each distinct color traces out a constant level on the normalized surface being drawn.

Although the class is intended to be used by other programs to display surfaces produced by those programs, the class has a main method making it possible to run it in a stand-alone mode for testing. When run as a stand-alone program, the class produces and displays six individual surfaces with the lowest point in the upper left corner and the highest point in the lower right corner. The scale strip is displayed immediately below each surface. The six surfaces

Listing 29. Source code for ImgMod29.java.

are stacked in the upper left corner of the screen. (You must physically move the ones on the top to see the ones on the bottom.) The stacking order of the surfaces from bottom to top is based on display types in the order 0, 1, 2, 3, 4, and 5. The surfaces that are displayed are 3D parabolas. Some of the surfaces show axes and some do not.

The constructor defines an anonymous inner class listener on the close button on the frame. Clicking the close button will terminate the program that uses an object of this class.

Tested using J2SE 5.0 and WinXP

```
*****/
import java.awt.*;
import java.awt.event.*;

class ImgMod29 extends Frame{
    int dataWidth;
    int dataHeight;
    int blockSize;
    boolean axis;
    double[][] data;

    ImgMod29(double[][] dataIn,int blockSize,
             boolean axis,int display){
        //Get and save several important values
        this.blockSize = blockSize;
        this.axis = axis;
        dataHeight = dataIn.length;
        dataWidth = dataIn[0].length;
        boolean logPlot = false;
        int displayType = display;
```

Listing 29. Source code for ImgMod29.java.

```
//plot types 0, 1, and 2 with no log
// conversion for display parameter
// value = 0, 1, or 2. This is the default
// and no special code is required.
//plot types 0, 1, and 2 with log conversion
// for display parameter value = 3, 4, or 5.
if(display == 3){
    displayType = 0;
    logPlot = true;
}else if(display == 4){
    displayType = 1;
    logPlot = true;
}else if(display == 5){
    displayType = 2;
    logPlot = true;
}else if((display > 5) || (display < 0)){
    System.out.println(
        "DisplayType input error, terminating");
    System.exit(0);
}//end if

//Make a copy of the input data array to
// avoid damaging the original data.
data = new double[dataHeight][dataWidth];
for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
        data[row][col] = dataIn[row][col];
    }//end loop on col
}//end loop on row

if(logPlot){//Convert to log base 10.
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            //Change the sign on negative values
            // before converting to log values.
```

Listing 29. Source code for ImgMod29.java.

```
        if(data[row][col] < 0){
            data[row][col] = -data[row][col];
        }//end if
        if(data[row][col] > 0){
            //Convert value to log base 10. Log
            // of 0 is undefined. Just leave it
            // at 0.
            data[row][col] =
                Math.log10(data[row][col]);
        }//end if
    }//end col loop
} //end row loop
} //end if on logPlot

//Force the data into the range from 0 to 255
// regardless of whether or not it has been
// converted to log values.
scaleTheSurfaceData();

Canvas surface = null;
Canvas scale = null;

//Establish the format based on the value of
// the parameter named display.
if(displayType == 0){
    //Create a type 0 Canvas object to draw the
    // surface on. This is a gray scale
    // display.
    surface = new CanvasType0surface();
    //Create a Canvas object to draw the scale
    // on.
    scale = new CanvasType0scale();
}else if(displayType == 1){
    //Color Shift plot
    surface = new CanvasType1surface();
    scale = new CanvasType1scale();
}
```

Listing 29. Source code for ImgMod29.java.

```
}else if(displayType == 2){
    //Color Contour plot.
    surface = new CanvasType2surface();
    scale = new CanvasType2scale();
};//end if-else on display type

//Add the plotted surface to center of the
// Frame
add(BorderLayout.CENTER,surface);
//Add the scale to bottom of Frame
add(BorderLayout.SOUTH,scale);
//Cause the size of the Frame to be just
// right to contain the two Canvas objects.
pack();

//Set Frame cosmetics and make it visible.
setTitle("Copyright 2005 R.G.Baldwin");
setVisible(true);

//Use an anonymous class to register a window
// listener on the Frame. This class extends
// WindowAdapter
addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e){
        System.exit(0);
    };//end windowClosing
});//end class definition
);//end addWindowListener

};//end constructor
//-----//

double min;
double max;
//This method is used to scale the surface data
// to force it to fit in the range from 0 to
```

Listing 29. Source code for ImgMod29.java.

```
// 255.
void scaleTheSurfaceData(){
    //Find the minimum surface value.
    min = Double.MAX_VALUE;
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            if(data[row][col] < min)
                min = data[row][col];
        }//end col loop
    }//end row loop

    //Shift all values up or down to force new
    // minimum value to be 0.
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            data[row][col] = data[row][col] - min;
        }//end col loop
    }//end row loop

    //Now get the maximum value of the shifted
    // surface values
    max = -Double.MAX_VALUE;
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            if(data[row][col] > max)
                max = data[row][col];
        }//end col loop
    }//end row loop

    //Now scale all values to cause the new
    // maximum value to be 255.
    for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
            data[row][col] =
                data[row][col] * 255/max;
        }//end col loop
    }
```

Listing 29. Source code for ImgMod29.java.

```
    }//end row loop
} //end scaleTheSurfaceData
//-----//

//main method for self-testing the class
public static void main(String[] args){
    //Create the array of test data.
    int numberRows = 59;
    int numberCols = 59;
    double[][] data =
        new double[numberRows][numberCols];
    int blockSize = 2;

    //Create a surface with a minimum at the
    // upper left corner and a maximum at the
    // lower right corner. This surface is
    // a 3D parabola.
    for(int row = 0; row < numberRows; row++){
        for(int col = 0; col < numberCols; col++){
            int xSquare = col * col;
            int ySquare = row * row;
            data[row][col] = xSquare + ySquare;
        } //end col loop
    } //end row loop

    //Instantiate objects to display the test
    // data surface in six different formats on
    // top of one another in the upper left
    // corner of the screen. Represent each
    // surface value by a colored square that is
    // blockSize pixels on each side. Draw a red
    // axis at the center of some of the
    // surfaces.
    new ImgMod29(data, blockSize, true, 0);
    new ImgMod29(data, blockSize, false, 1);
    new ImgMod29(data, blockSize, true, 2);
```


Listing 29. Source code for ImgMod29.java.

```
    new ImgMod29(data, blockSize, true, 3);
    new ImgMod29(data, blockSize, false, 4);
    new ImgMod29(data, blockSize, true, 5);
} //end main
//-----//

int horizCenter;
int vertCenter;
//This helper method is used to find the
// horizontal and vertical center of the
// surface. These values are used to locate
// the red axes that are drawn on the surface.
// Note that the returned values depend on
// whether the dimensions of the surface are
// odd or even.
void getCenter(){
    if(dataWidth%2 == 0){ //even
        horizCenter =
            dataWidth * blockSize/2 + blockSize/2;
    }else{ //odd
        horizCenter = dataWidth * blockSize/2;
    } //end else

    if(dataHeight%2 == 0){ //even
        vertCenter =
            dataHeight * blockSize/2 + blockSize/2;
    }else{ //odd
        vertCenter = dataHeight * blockSize/2;
    } //end else
} //end getCenter
//-----//

//Note that the following six classes are
// inner classes. This makes it possible for
// methods in the class to access instance
// variables and methods of the containing
```

Listing 29. Source code for ImgMod29.java.

```
// object.

//This class is used to draw a gray scale
// surface ranging from white at the high end
// to black at the low end with a smooth
// gradient in between.
class CanvasType0surface extends Canvas{
    CanvasType0surface(){//constructor
        //Set the size of the Canvas based on the
        // size of the surface and the size of the
        // square used to represent each value on
        // the surface.
        setSize(dataWidth * blockSize,
                dataHeight * blockSize);
        getCenter();
    }//end constructor

    //Override the paint method to draw the
    // surface.
    public void paint(Graphics g){
        //Vary from white to black going from high
        // to low.
        Color color = null;
        for(int row = 0;row < dataHeight;row++){
            for(int col = 0;col < dataWidth;col++){
                //Add in red, green, and blue in
                // proportion to the value of the
                // surface height.
                int red = (int)data[row][col];
                int green = red;
                int blue = red;
                //Compute the color value for the
                // point on the surface.
                color = new Color(red,green,blue);
                //Set the color value.
                g.setColor(color);
            }
        }
    }
}
```

Listing 29. Source code for ImgMod29.java.

```
        //Draw a square of the specified size
        //in the specified color at the
        // specified location.
        g.fillRect(col * blockSize,
                    row * blockSize,
                    blockSize,
                    blockSize);
    } //end col loop
} //end row loop

//If axis is true, draw red lines to form
// an origin at the center
if(axis){
    g.setColor(Color.RED);
    g.drawLine(0,vertCenter,2*horizCenter,
                vertCenter);
    g.drawLine(horizCenter,0,horizCenter,
                2*vertCenter);
} //end if
} //end paint
} //end inner class CanvasType0surface
//=====//

//Note that this is an inner class.
//This class is used to construct a color scale
// that matches the color scheme used in the
// class named CanvasType0surface.
class CanvasType0scale extends Canvas{
    //Set the physical height of the scale strip
    // in pixels.
    int scaleHeight = 6 * blockSize;

    CanvasType0scale(){ //constructor
        //Set the size of the Canvas based on the
        // width of the surface and the size of the
        // square used to represent each value on
```

Listing 29. Source code for ImgMod29.java.

```
// the surface.
setSize(dataWidth * blockSize,scaleHeight);
} //end constructor

//Override the paint method to draw the
// scale strip.
public void paint(Graphics g){
    //Vary from white to black going from 255
    // to 0.
    Color color = null;
    //Don't draw in top row. Leave it blank to
    // separate the scale strip from the
    // drawing of the surface above it.
    for(int row = 1;row < scaleHeight;row++){
        for(int col = 0;col < dataWidth;col++){

            //Compute the value of the scale
            // surface.
            int scaleValue = 255 * col /
                                (dataWidth - 1);

            //See the class named
            // CanvasType0surface for explanatory
            // comments regarding the following
            // color algorithm.
            int red = scaleValue;
            int green = red;
            int blue = red;
            color = new Color(red,green,blue);
            g.setColor(color);
            g.fillRect(col * blockSize,
                        row * blockSize,
                        blockSize,
                        blockSize);
        } //end col loop
    } //end row loop
}
```

Listing 29. Source code for ImgMod29.java.

```
    }//end paint

//end inner class CanvasType0scale
//=====//

//This class is used to draw a surface with the
// colors ranging from blue at the low end
// through aqua, green, and yellow to red at
// the high end with a smooth gradient from 1
// to 254. The lowest point with a value of 0
// is colored black. The highest point with a
// value of 255 is colored white.
class CanvasType1surface extends Canvas{

    CanvasType1surface(){//constructor
        //Set the size of the Canvas based on the
        // size of the surface and the size of the
        // square used to represent each value on
        // the surface.
        setSize(dataWidth * blockSize,
                dataHeight * blockSize);
        getCenter();
    }//end constructor

    //Override the paint method to draw the
    // surface.
    public void paint(Graphics g){
        //Vary color as described in the comments
        // above.
        Color color = null;
        for(int row = 0; row < dataHeight; row++){
            for(int col = 0; col < dataWidth; col++){
                int red = 0;
                int green = 0;
                int blue = 0;
```

Listing 29. Source code for ImgMod29.java.

```
if((int)data[row][col] == 255){
    red = green = blue = 255;//white
}else if((int)data[row][col] == 0 ){
    red = green = blue = 0;//black

}else if(((int)data[row][col] > 0) &&
        ((int)data[row][col] <= 63)){
    int temp = 4 * ((int)data[row][col]
                    - 0);

    blue = 255;
    green = temp;

}else if(((int)data[row][col] > 63) &&
        ((int)data[row][col] <= 127)){
    int temp = 4 * ((int)data[row][col]
                    - 64);

    green = 255;
    blue = 255 - temp;

}else if(((int)data[row][col] > 127) &&
        ((int)data[row][col] <= 191)){
    int temp = 4 * ((int)data[row][col]
                    - 128);

    green = 255;
    red = temp;

}else if(((int)data[row][col] > 191) &&
        ((int)data[row][col] <= 254)){
    int temp = 4 * ((int)data[row][col]
                    - 192);

    red = 255;
    green = 255 - temp;

}else{//impossible condition
    System.out.println(
        "Should not reach here.");
```

Listing 29. Source code for ImgMod29.java.

```
        System.exit(0);
    }//end else

    //Compute the color value for the
    // point on the surface.
    color = new Color(red,green,blue);
    //Set the color value.
    g.setColor(color);
    //Draw a square of the specified size
    // in the specified color at the
    // specified location.
    g.fillRect(col * blockSize,
               row * blockSize,
               blockSize,
               blockSize);
    }//end col loop
} //end row loop

//If axis is true, draw red lines to form
// an origin at the center
if(axis){
    g.setColor(Color.RED);
    g.drawLine(0,vertCenter,2*horizCenter,
               vertCenter);
    g.drawLine(horizCenter,0,horizCenter,
               2*vertCenter);
} //end if
} //end paint
} //end inner class CanvasType1surface
//=====//

//Note that this is an inner class. This class
// is used to construct a color scale that
// matches the color scheme used in the class
// named CanvasType1surface.
class CanvasType1scale extends Canvas{
```

Listing 29. Source code for ImgMod29.java.

```
int scaleHeight = 6 * blockSize;

CanvasType1scale(){//constructor
    //Set the size of the Canvas based on the
    // width of the surface and the size of the
    // square used to represent each value on
    // the surface.
    setSize(dataWidth * blockSize,scaleHeight);
}//end constructor

//Override the paint method to draw the
// scale.
public void paint(Graphics g){
    //Vary from yellow to blue going from 255
    // to 0.
    Color color = null;
    for(int row = 1;row < scaleHeight;row++){
        for(int col = 0;col < dataWidth;col++){

            int scaleValue = 255 * col/(
                                dataWidth - 1);

            // See the class named
            // CanvasType1surface for explanatory
            // comments regarding this color
            // algorithm.
            int red = 0;
            int green = 0;
            int blue = 0;
            if(scaleValue == 255){
                red = green = blue = 255;//white
            }else if(scaleValue == 0 ){
                red = green = blue = 0;//black

            }else if((scaleValue > 0) &&
                    (scaleValue <= 63)){
                scaleValue = 4 * (scaleValue - 0);
```


Listing 29. Source code for ImgMod29.java.

```
        blue = 255;
        green = scaleValue;

    }else if((scaleValue > 63) &&
            (scaleValue <= 127)){
        scaleValue = 4 * (scaleValue - 64);
        green = 255;
        blue = 255 - scaleValue;

    }else if((scaleValue > 127) &&
            (scaleValue <= 191)){
        scaleValue = 4 * (scaleValue - 128);
        green = 255;
        red = scaleValue;

    }else if((scaleValue > 191) &&
            (scaleValue <= 254)){
        scaleValue = 4 * (scaleValue - 192);
        red = 255;
        green = 255 - scaleValue;

    }else{//impossible condition
        System.out.println(
            "Should not reach here.");
        System.exit(0);
    }//end else

    color = new Color(red,green,blue);
    g.setColor(color);
    g.fillRect(col * blockSize,
                row * blockSize,
                blockSize,
                blockSize);
    }//end col loop
} //end row loop
} //end paint
```

Listing 29. Source code for ImgMod29.java.

```
//end inner class CanvasType1scale
//=====//

//This is a utility method used by the two
// inner classes that follow. The purpose of
// this method is to establish a color palette
// containing 23 distinct Colors and shades of
// gray. The values shown in comments
// represent the values of red, green, and blue
// for that specific color.
Color[] getColorPalette(){
    //Note that the following is an initialized
    // 1D array of type Color.
    Color[] colorPalette = {
        Color.BLACK,//          0,  0,  0
        Color.GRAY,//          128,128,128
        Color.LIGHT_GRAY,//      192,192,192
        Color.BLUE,//           0,  0,255
        new Color(100,100,255),//100,100,255
        new Color(140,140,255),//140,140,255
        new Color(175,175,255),//175,175,255
        Color.CYAN,//           0,255,255
        new Color(140,255,255),//140,255,255
        Color.GREEN,//          0,255,  0
        new Color(140,255,140),//140,255,140
        new Color(200,255,200),//200,255,200
        Color.PINK,//           255,175,175
        new Color(255,140,255),//255,140,255
        Color.MAGENTA,//        255,  0,255
        new Color(255,0,140),    //255,  0,140
        Color.RED,//            255,  0,  0
        new Color(255,100,0),//    255,100,  0
        Color.ORANGE,//          255,200,  0
        new Color(255,225,0),//    255,225,  0
        Color.YELLOW,//          255,255,  0
    }
```

Listing 29. Source code for ImgMod29.java.

```
        new Color(255,255,150), //255,255,150
        Color.WHITE}); //          255,255,255

    return colorPalette;
} //end getColorPalette
//=====//

//Note that this is an inner class.
//This class is used to draw a surface
// representing the heights of the points on
// the surface using the colors and shades of
// gray defined in the color palette..
class CanvasType2surface extends Canvas{

    CanvasType2surface(){//constructor
        //Set the size of the Canvas based on the
        // size of the surface and the size of the
        // square used to represent each value on
        // the surface.
        setSize(dataWidth * blockSize,
                dataHeight * blockSize);
        getCenter();
    } //end constructor

    //Override the paint method to draw the
    // surface.
    public void paint(Graphics g){
        Color[] colorPalette = getColorPalette();

        for(int row = 0; row < dataHeight; row++){
            for(int col = 0; col < dataWidth; col++){
                //Quantize the surface into a set of
                // levels where the number of levels is
                // equal to the number of colors in the
                // color palette.
                int quantizedData = (int)(Math.round(
```

Listing 29. Source code for ImgMod29.java.

```
                data[row][col]*(
                    colorPalette.length-1)/255));
        //Set the color for this point to the
        // corresponding color from the
        // palette by matching the integer
        // value of the level and the index
        // value of the palette.
        g.setColor(colorPalette[
                                quantizedData]);
        //Draw a square in the output image of
        // the specified color at the specified
        // location.
        g.fillRect(col * blockSize,
                    row * blockSize,
                    blockSize,
                    blockSize);
    }//end col loop
} //end row loop

//If axis is true, draw red lines to form
// an origin at the center
if(axis){
    g.setColor(Color.RED);
    g.drawLine(0,vertCenter,2*horizCenter,
                                vertCenter);
    g.drawLine(horizCenter,0,horizCenter,
                                2*vertCenter);
} //end if
} //end paint
} //end inner class CanvasType2surface
//=====//

//Note that this is an inner class. This class
// is used to construct a color scale that
// matches the color scheme used in the class
// named CanvasType2surface.
```

Listing 29. Source code for ImgMod29.java.

```
class CanvasType2scale extends Canvas{
    int scaleHeight = 6 * blockSize;

    CanvasType2scale(){//constructor
        //Set the size of the Canvas based on the
        // width of the surface and the size of the
        // square used to represent each value on
        // the surface.
        setSize(dataWidth * blockSize,scaleHeight);
    }//end constructor

    //Override the paint method to draw the
    // scale.
    public void paint(Graphics g){
        Color[] colorPalette = getColorPalette();

        for(int row = 1;row < scaleHeight;row++){
            for(int col = 0;col < dataWidth;col++){

                //Get the value of the point on the
                // scale surface.
                double scaleValue =
                    255.0 * col/dataWidth;
                //See the class named
                // CanvasType2surface for an
                // explanation of this color
                // algorithm.
                int quantizedData = (int)(Math.round(
                    scaleValue*(
                        colorPalette.length-1)/255));
                g.setColor(colorPalette[
                    quantizedData]);
                g.fillRect(col * blockSize,
                    row * blockSize,
                    blockSize,
                    blockSize);
            }
        }
    }
}
```

Listing 29. Source code for ImgMod29.java.

```
        }//end col loop
    }//end row loop
} //end paint

} //end inner class CanvasType2scale
//=====//

} //end outer class ImgMod29
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1489-Plotting 3D Surfaces using Java
- File: Java1489.htm
- Published: 05/31/05

Learn how to write a Java class that uses color to plot 3D surfaces in six different formats and a wide range of sizes. The class is extremely easy to use. You can incorporate the 3D plotting capability into your own programs by inserting a single statement into your programs.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1492-Plotting Large Quantities of Data using Java
Learn how to use Java to plot millions of multi-channel data values in an easy-to-view format with very little programming effort.

Revised: Fri Oct 16 23:15:53 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
 - [Sample output for PlotALot01 class](#)
 - [Usage information](#)
 - [Plotting format](#)
 - [On to the next page](#)
 - [Sample output for PlotALot02 class](#)
 - [Superimposed data](#)
 - [Sample output for PlotALot03 class](#)
 - [Same data, two colors](#)
 - [Sample output for PlotALot04 class](#)

- [Same data, three colors](#)
- [Sample programs](#)
 - [The class named PlotALot01](#)
 - [Purpose of the class](#)
 - [Usage information](#)
 - [Different plotting objects](#)
 - [Beginning of the class named PlotALot01](#)
 - [Feed the plotting object titled "A"](#)
 - [Plot the data](#)
 - [Feed and plot the object titled "B"](#)
 - [Some instance variables](#)
 - [The first overloaded constructor](#)
 - [Save the parameter values](#)
 - [A temporary Page object](#)
 - [Display some information](#)
 - [Dispose of the temporary Page object](#)
 - [Compute and display the remaining plotting parameters](#)
 - [Instantiate first usable Page object](#)
 - [The other overloaded constructor](#)
 - [The feedData method](#)
 - [The MyCanvas class, a preview](#)
 - [The plotData method](#)
 - [Make all pages invisible](#)
 - [Make the pages visible in reverse order](#)
 - [The other overloaded version of the plotData method](#)
 - [The Page class](#)
 - [An anonymous terminator for the Page class](#)
 - [The putData method of the Page class](#)
 - [The MyCanvas class](#)
 - [The overridden paint method](#)
 - [Plot the points](#)
 - [Draw an oval](#)
 - [Connect the points with straight lines](#)
 - [End of the PlotALot01 class](#)
 - [The class named PlotALot02](#)

- [Designed for two-channel data](#)
- [The class named PlotALot02 and the main method](#)
- [The feedData method](#)
- [The putData method](#)
- [The MyCanvas class](#)
- [The overridden paint method](#)
- [New code in the overridden paint method](#)
- [The class named PlotALot03](#)
 - [Two-channel data on alternating axes](#)
 - [Modified constructor code](#)
 - [The overridden paint method](#)
- [The class named PlotALot04](#)
 - [Three steps for using the class](#)
- [Run the programs](#)
- [Summary](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

In one of my earlier modules titled [Plotting Engineering and Scientific Data using Java](#), I published a generalized 2D plotting class that makes it easy to cause other programs to display their outputs in 2D [Cartesian coordinates](#). I have used that plotting class in numerous modules since I published it. Hopefully, some of you have been using it as well.

In another of my earlier modules titled [Plotting 3D Surfaces using Java](#), I presented and explained a 3D surface plotting class that is also very easy to use. I have used that class in several modules since then, and I will be using it in many future modules as well.

Plotting large quantities of data

One of the common requirements of engineering, technical, and scientific computing is to be able to plot and to examine very large quantities of data. This is particularly true in time-series analysis, spectral analysis, and digital signal processing ([DSP](#)). I will present and explain four separate Java classes in this module, which make it very easy to plot and to examine large quantities of data in Java programs.

How do you use these classes?

All that's necessary to use these classes to plot large quantities of data is to:

1. Instantiate a plotting object of type **PlotALot01** , **PlotALot02** , **PlotALot03** or **PlotALot04** .
2. Feed the data values that need to be plotted to the plotting object as they become available.
3. Call a method named **plotData** on the plotting object when all of the data has been fed to the object.

It couldn't be easier

The choice among the four classes listed above depends on whether you need to plot one, two, or three channels of data, and the format in which you want to plot the data. The class named **PlotALot01** is used to plot single-channel data. The classes named **PlotALot02** and **PlotALot03** are used to plot two-channel data in two different formats. The class named **PlotALot04** is used to plot three-channel data.

A free plotting class

If you arrived at this page seeking a free Java class for plotting your data, you are in luck. Just copy the source code for the classes in [Listing 35](#) through [Listing 38](#) near the end of this module and feel free to use them as described in the comments in the source code.

On the other hand, if you would like to learn how the classes do what they do, and perhaps use your programming skills to improve them, keep reading. Hopefully, once you have finished the module, you will have learned quite a lot about plotting large quantities of data using Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Sample output for PlotALot01 class.
- [Figure 2](#). Sample output for PlotALot02 class.
- [Figure 3](#). Sample output for PlotALot03 class.
- [Figure 4](#). Sample output for PlotALot04 class.
- [Figure 5](#). Self-test output for PlotALot01.

Listings

- [Listing 1](#). Beginning of the class named PlotALot01.
- [Listing 2](#). Feed the plotting object titled "A".
- [Listing 3](#). Plot the data.
- [Listing 4](#). Feed and plot the object titled "B".
- [Listing 5](#). Some instance variables.
- [Listing 6](#). The first overloaded constructor.
- [Listing 7](#). Save the parameter values.
- [Listing 8](#). A temporary Page object.
- [Listing 9](#). Display some information.
- [Listing 10](#). Dispose of the temporary Page object.
- [Listing 11](#). Compute and display the remaining plotting parameters.
- [Listing 12](#). Instantiate first usable Page object.
- [Listing 13](#). The other overloaded constructor.
- [Listing 14](#). The feedData method.
- [Listing 15](#). Beginning of the plotData method.
- [Listing 16](#). Make all pages invisible.
- [Listing 17](#). Make the pages visible in reverse order.
- [Listing 18](#). The other overloaded version of the plotData method.
- [Listing 19](#). Beginning of the class named Page.
- [Listing 20](#). An anonymous terminator for the Page class.

- [Listing 21](#). The putData method of the Page class.
- [Listing 22](#). Beginning of the MyCanvas class.
- [Listing 23](#). Beginning of the overridden paint method.
- [Listing 24](#). Beginning of code to plot the points.
- [Listing 25](#). Draw an oval.
- [Listing 26](#). Connect the points with straight lines.
- [Listing 27](#). The class named PlotALot02 and the main method.
- [Listing 28](#). The feedData method.
- [Listing 29](#). The putData method.
- [Listing 30](#). Beginning of the MyCanvas class.
- [Listing 31](#). Beginning of the overridden paint method.
- [Listing 32](#). New code in the overridden paint method.
- [Listing 33](#). Modified constructor code.
- [Listing 34](#). The overridden paint method.
- [Listing 35](#). PlotALot01.java.
- [Listing 36](#). PlotALot02.java.
- [Listing 37](#). PlotALot03.java.
- [Listing 38](#). PlotALot04.java.

Preview

The four classes that I will present and explain in this module are designed to make it easy for you to plot and examine large quantities of data.

Sample output for PlotALot01 class

The first class named **PlotALot01** is designed for the plotting of large quantities of single-channel data. [Figure 1](#) shows an example of the plotted output from this class when used to plot a small amount of data.

Figure 1. Sample output for PlotALot01 class.

Figure 1. Sample output for PlotALot01 class.

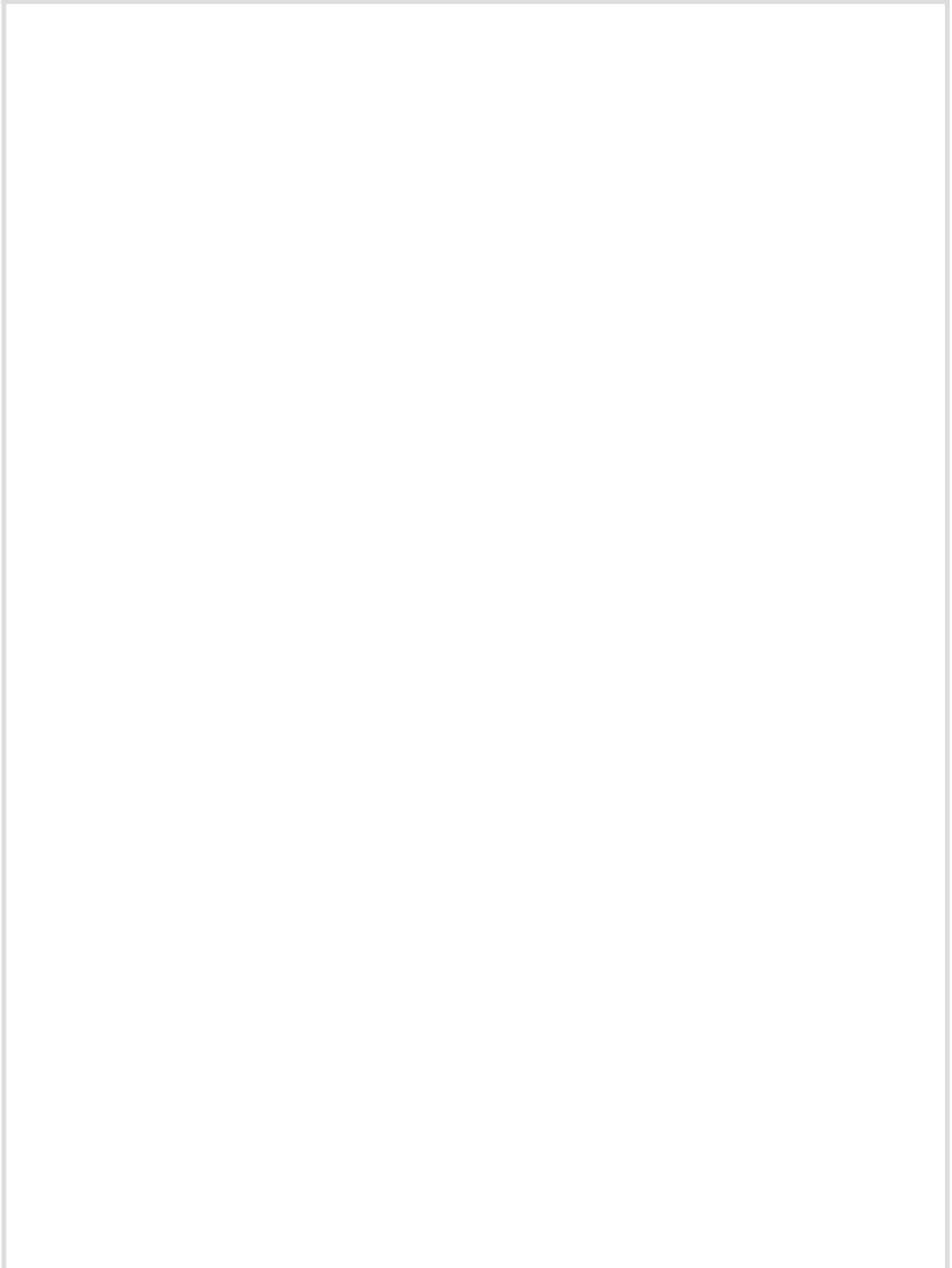
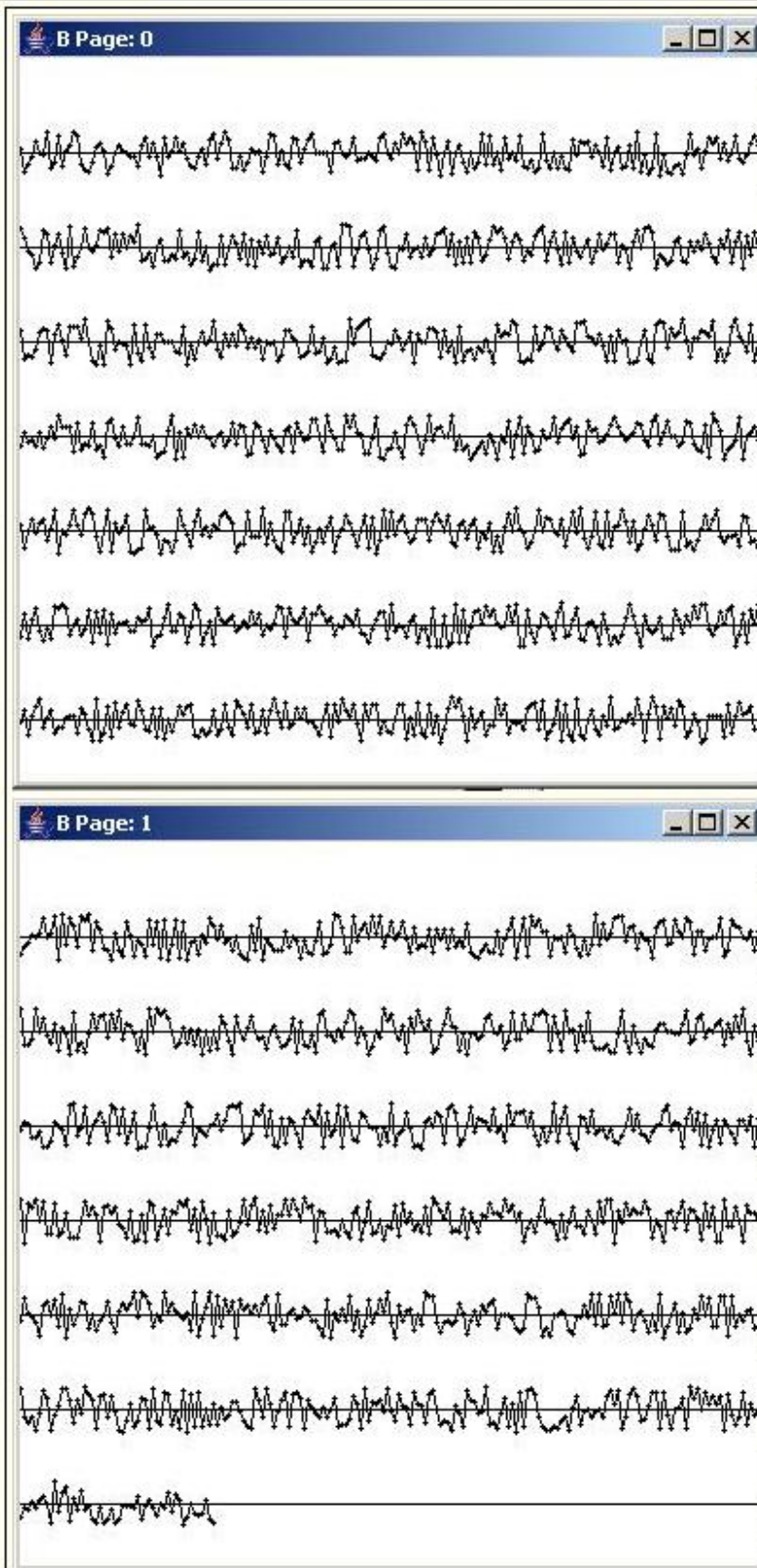


Figure 1. Sample output for PlotALot01 class.



Usage information

To use the class named **PlotALot01** to plot data, you first instantiate an object of the class, and then you feed data to it as the data becomes available within your program.

When all of the data has been fed to the plotting object, you call a method named **plotData** on the object. This causes the object to produce one or more pages of plotted data in a stack on the screen. The page containing the earliest data is on the top of the stack and the page containing the latest data on the bottom of the stack.

Plotting format

The first data sample is plotted on the left end of the top trace on the top page (*titled Page: 0*) . Successive data values are plotted from left to right across the page. When the data for the first trace reaches the right end of the trace, the next data sample is plotted at the left end of a new trace that is created below the current trace. Hence, the chronological order of the data is from left to right, top to bottom.

A horizontal axis is drawn for each trace. Positive data values are plotted above the axis and negative values are plotted below the axis.

On to the next page

When the bottom trace on a page is filled, a new page is created automatically. The next data sample is plotted on the left end of the top trace on the new page and the process described above is repeated until that page also become full. Then a new page is created, etc.

Nearly unlimited plotting capacity

You can cause the page size to be as large as you want up to the full size of the screen on your computer. You can create as many pages as you want and you can place as many traces on each page as you want.

Other than the amount of memory that is available to the Java virtual machine, *(and perhaps some limit on the number of **Page** objects allowed by the operating system)*, there is almost no limit to the number of pages that can be produced and the amount of data that can be plotted.

Millions of data values plotted

I have successfully plotted two million data values in 141 full screen pages on a modest laptop computer with no difficulty whatsoever. When I pushed that total up to eight million data values in 563 full screen pages, the plotting process slowed down, but I was still able to display and examine the plots. The practical limit on my computer seems to be somewhere between two million and eight million data values.

Two sample pages

[Figure 1](#) shows two pages that were physically removed from the stack and arranged with the page containing the earliest data above the page containing the latest data for publication in this module.

Two overloaded constructors

Two overloaded constructors are provided for the class. One constructor plots the data using a set of default plotting parameters. This constructor is provided for extreme ease of use. The only information that you must provide to this constructor is a string that becomes part of the title for each page.

(The pages in [Figure 1](#) were plotted using default plotting parameters with a title string of "B". The amount of data that was

fed to the plotting object for [Figure 1](#) filled Page 0 and almost filled Page 1.)

Can change default plotting parameters

I coded the values of the default plotting parameters to make the results suitable for use in this narrow publication format. If you don't like my choice of default plotting parameters, you can change them to values that you find more useful. For example, you could cause the default size of the **Frame** object to fill your screen, allowing you to plot quite a lot of data on each page.

Control over the plotting parameters

The other overloaded constructor takes seven parameters that allow you to control all aspects of the plotting format including:

- Page title
- Frame width and hence plotted data width
- Frame height, spacing between traces, and hence the number of traces per page
- Spacing between samples, number of traces per page, and hence the number of samples per page
- Width and height of an oval that is used to mark each sample on the plot

The plotted sample values are connected by a straight line. Each sample is marked with an oval. You can specify the width and the height of the oval in pixels. If you set the width and height to zero, the oval simply disappears from the plot.

Default plotting parameters in [Figure 1](#)

The plots in [Figure 1](#) were produced using the constructor that applies default plotting parameters. For example, the data was plotted using the default value

of two pixels per sample. Hence the lines connecting the sample values in [Figure 1](#) are very short.

The ovals in [Figure 1](#) had a default width and height of two pixels each. At this small size, the ovals end up looking more like plus characters than ovals.

The overall parameters governing the plot in [Figure 1](#) were:

Title: B
Frame width: 400
Frame height: 410
Page width: 392
Page height: 383
Trace spacing: 50
Sample spacing: 2
Traces per page: 7
Samples per page: 1372

Explanation of terms

Some explanation of the terminology in the above list is probably in order. The **Frame width** and **Frame height** are the actual width and height of the **Frame** objects shown in [Figure 1](#).

The **Page width** and **Page height** are the width and height of a **Canvas** object contained in the **Frame** object, upon which the plotting is performed. The width of the **Canvas** actually controls the number of samples that can be plotted in each trace.

The **Trace spacing** is the number of pixels that separate each of the horizontal axes on a page in [Figure 1](#).

The **Sample spacing** specifies the number of pixels that are dedicated to each sample horizontally. In [Figure 1](#), that value is 2. This means that every other black pixel in [Figure 1](#) indicates the value of a data sample. The pixels in between are fillers.

The **Traces per page** specifies the number of horizontal axes on each page against which the data values are plotted.

The **Samples per page** gives the actual number of data values that are plotted on each page. This is determined from the values of **Traces per page** , **Sample spacing** , and **Page width** .

Location of the stack of plots

There are also two overloaded versions of the method named **plotData** . One version lets you specify the location of the upper left corner of the stack of pages relative to the upper left corner of the screen. The other version simply places the stack of pages in the upper left corner of the screen by default.

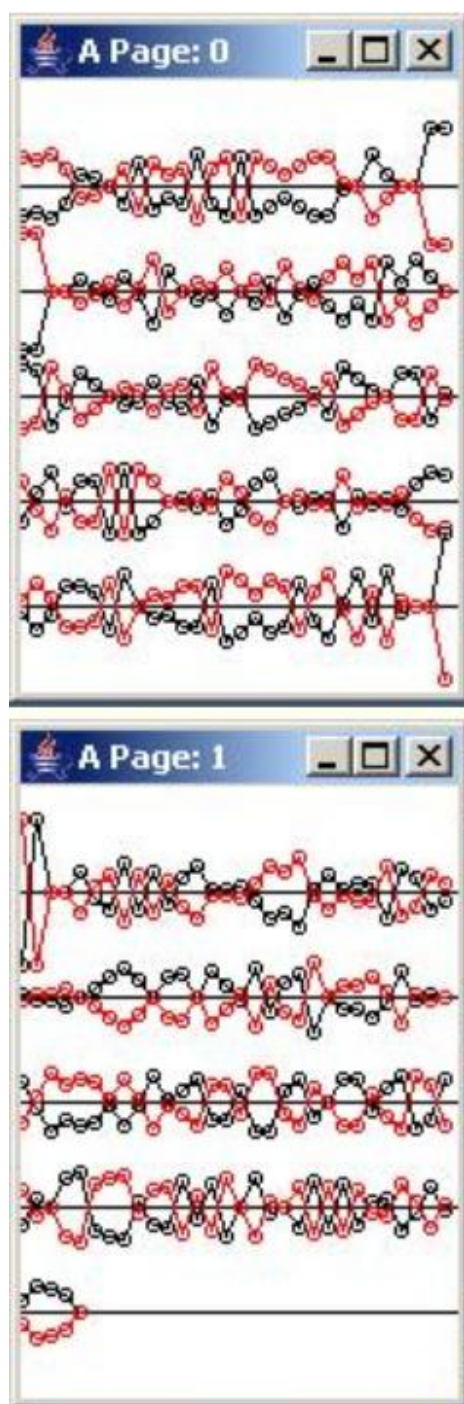
Sample output for PlotALot02 class

The class named **PlotALot01** is designed for the plotting of large quantities of data from a single channel as described above. The classes named **PlotALot02** and **PlotALot03** are each designed to plot two channels of data. These two classes plot the two-channel data in different formats.

Superimposed data

The class named **PlotALot02** provides all of the features described above for the class named **PlotALot01** , such as overloaded constructors, overloaded **plotData** methods, etc. In addition, it provides the capability to superimpose two sets of data on the same axes with one set being plotted in black and the other being plotted in red. This is illustrated in [Figure 2](#).

Figure 2. Sample output for PlotALot02 class.



Same data, different sign

The plots in [Figure 2](#) were produced by plotting two versions of the same data. The algebraic sign of each of the data values was inverted in one set of data relative to the other. Thus, the red plot in [Figure 2](#) is an upside down version of the black plot. This makes it easy to confirm that both plotting processes are behaving the same way.

Plotting parameters were controlled

The plots in [Figure 2](#) were produced using the version of the constructor that allows the user to control the plotting parameters. The overall plotting parameters for [Figure 2](#) are shown below:

Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
Traces per page: 5
Samples per page: 150

Larger ovals

As you can see, the ovals that were used to mark the sample values in [Figure 2](#) were larger than in [Figure 1](#). With a height and a width of four pixels, each oval turned out to be a circle centered on the sample value.

Horizontal scaling was greater

Also, the horizontal scaling in [Figure 2](#) was five pixels per sample as opposed to two pixels per sample in [Figure 1](#). As a result, the circles marking the samples were further apart, and the straight lines connecting the circles are often visible.

Sample output for PlotALot03 class

The classes named **PlotALot02** and **PlotALot03** are each designed to plot two channels of data. These two classes plot the two-channel data in different formats. Whereas **PlotALot02** superimposes the two sets of data on the same horizontal axes using color to provide visual separation, **PlotALot03** plots the two sets of data on alternating horizontal axes as shown in [Figure 3](#).

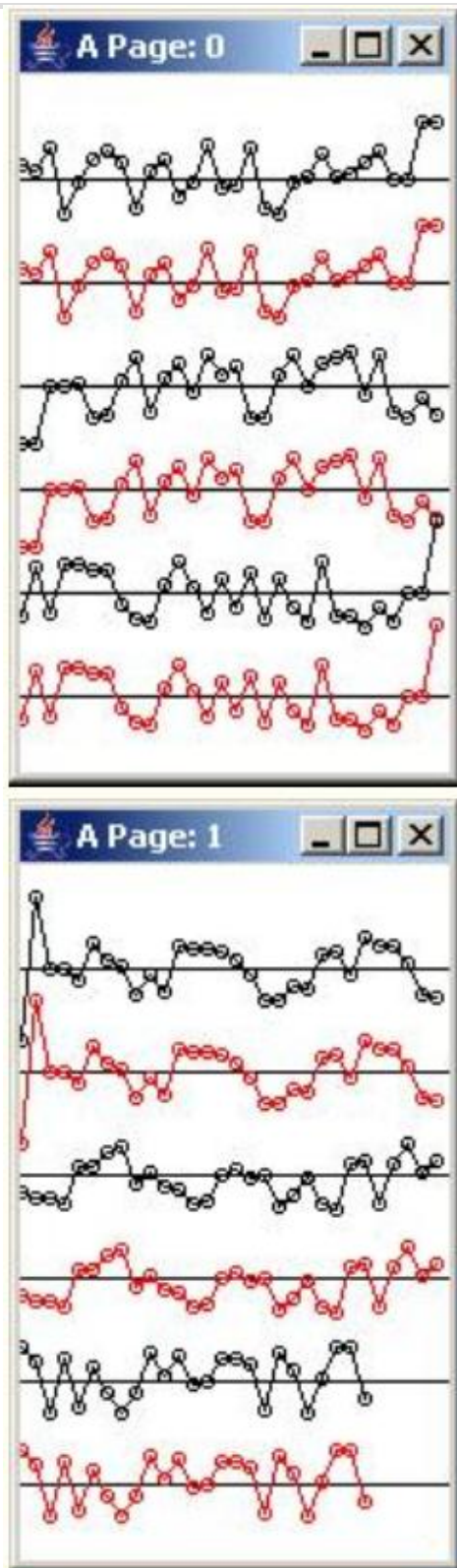
PlotALot03 also uses color to provide visual separation between the two sets of data. One set is plotted on the odd numbered axes in black. The other set is plotted on the even numbered axes in red.

The class named **PlotALot03** also provides all of the general capabilities described earlier for the class named **PlotALot01** that are appropriate for a two-channel plotting system.

Figure 3. Sample output for PlotALot03 class.



Figure 3. Sample output for PlotALot03 class.



Same data, two colors

The two sets of data plotted in [Figure 3](#) consisted of exactly the same values. Thus, the plots on the even numbered axes look just like the plots on the odd numbered axes except that one plot is red and the other is black. Using the same values for each set of data makes it easy to confirm that both plotting processes are behaving the same way.

The plotting parameters

The overall plotting parameters for [Figure 3](#) are shown below:

Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 90

Because **PlotALot03** doesn't superimpose the two sets of data, twice as many pages would be required for **PlotALot03** to plot a given amount of data as would be required by **PlotALot02** for the same Page size.

PlotALot03 will refuse to plot data for a set of plotting parameters that result in an odd number of traces on the page.

Sample output for PlotALot04 class

The class named **PlotALot04** plots three sets of data on separate horizontal axes as shown in [Figure 4](#). The first set of data is plotted in black. The second

set of data is plotted in red. The third set of data is plotted in blue. This class is particularly useful for displaying the input, output, and error signals involved in adaptive signal processing, for example.

The class named **PlotALot04** also provides all of the general capabilities described earlier for the class named **PlotALot01** that are appropriate for a three-channel plotting system.

Figure 4. Sample output for PlotALot04 class.

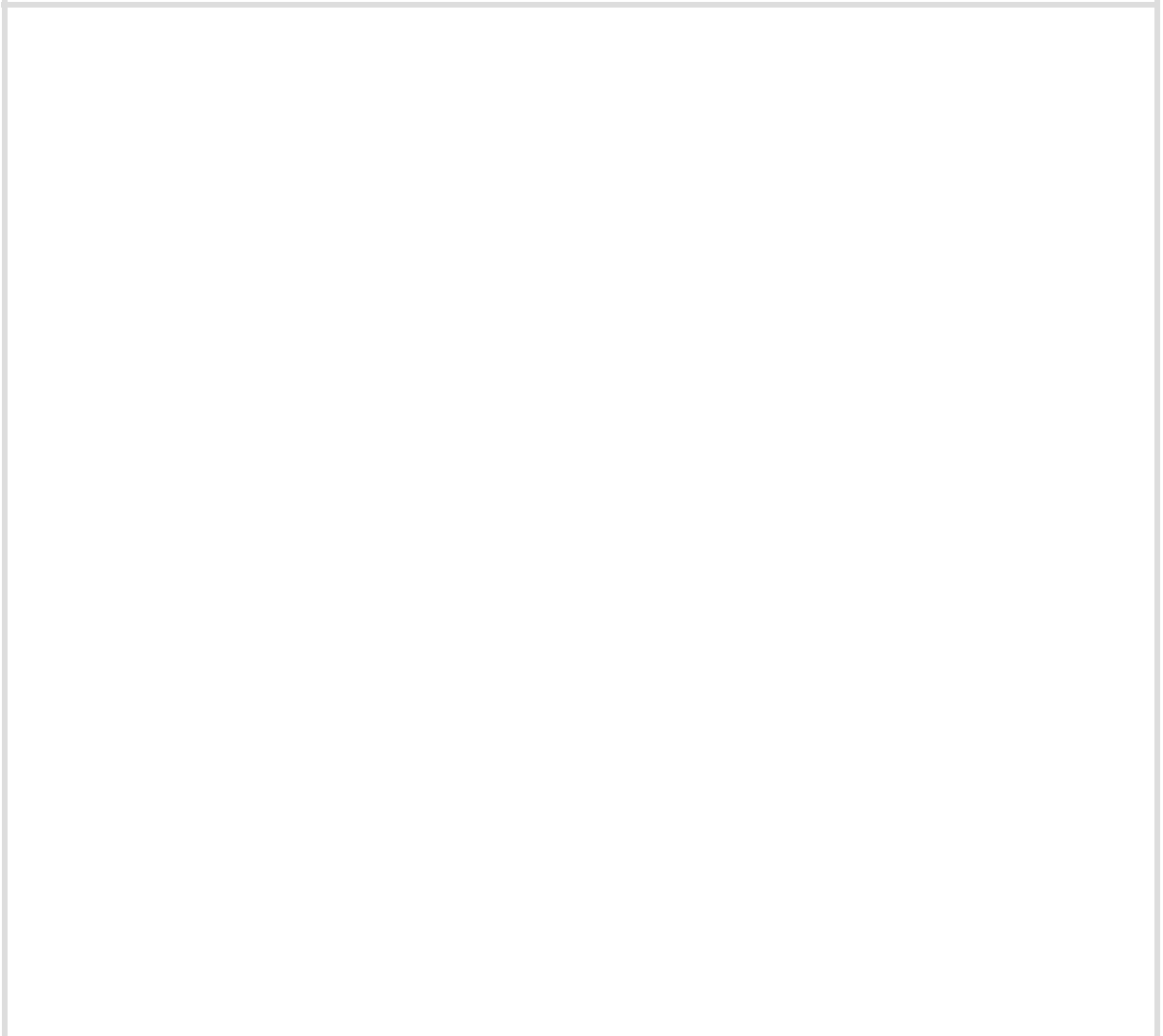
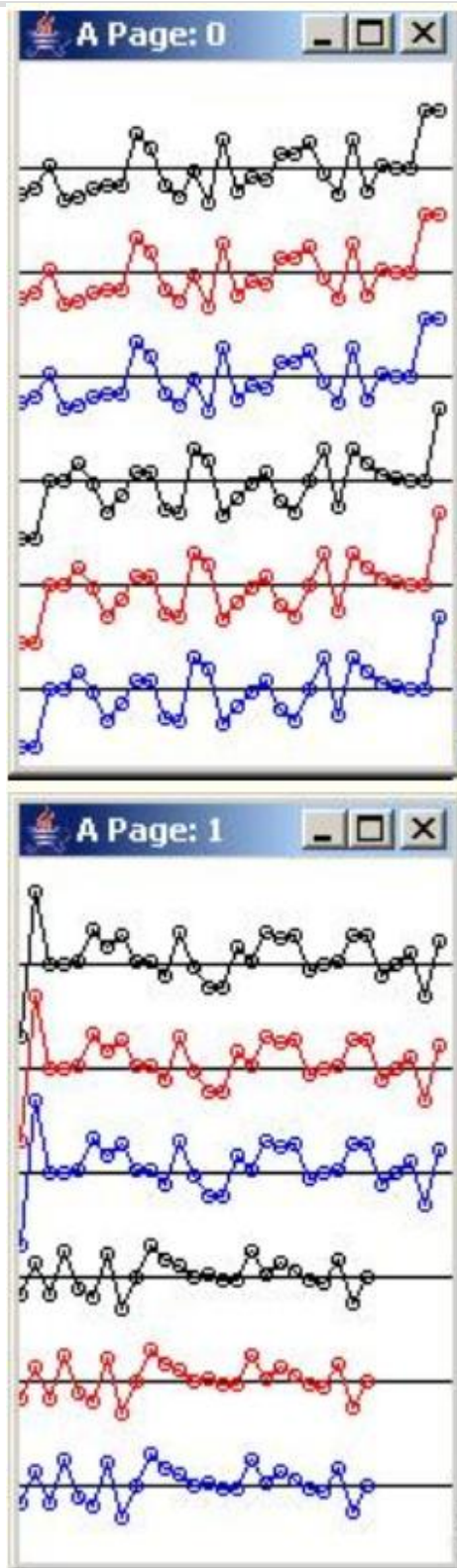


Figure 4. Sample output for PlotALot04 class.



Same data, three colors

The three sets of data plotted in [Figure 4](#) consisted of exactly the same values. Thus, the plots on the three different axes look just alike except that the first plot is black, the second plot is red and the third is blue. Using the same values for each set of data makes it easy to confirm that all three plotting processes are behaving the same way.

The plotting parameters

The overall plotting parameters for [Figure 4](#) are shown below:

Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 60

PlotALot04 will terminate if the number of traces per page is not evenly divisible by 3

Sample programs

The class named PlotALot01

Now that you know where we are heading, it's time to examine these four classes in detail. I will begin with the class named **PlotALot01** .

Purpose of the class

This class is designed to plot large amounts of data for a single channel. The class is particularly useful for plotting time series data. Also, by carefully adjusting the plotting parameters, this class can be used to plot large quantities of spectral data in a *waterfall* display with each new spectral estimate being plotted immediately below the previous estimate.

Usage information

The class provides a **main** method so that the class can be run as an application to test itself. The **main** method also illustrates how to use the class.

There are three steps involved in the use of this class for plotting large quantities of data:

1. Instantiate a plotting object of type **PlotALot01** using one of two overloaded constructors.
2. Feed the data that is to be plotted to the plotting object by calling the **feedData** method once for each data value.
3. Call one of two overloaded **plotData** methods on the plotting object once all of the data has been fed to the object. This causes all of the data to be plotted and causes the pages to be stacked in a particular location on the screen with page 0 on the top of the stack.

Different plotting objects

A program that uses this class for plotting can instantiate as many different plotting objects as are needed to plot all of the different sets of data that need to be plotted independently of one another.

(For example, a program that uses this class could instantiate one plotting object to plot time series data and a different plotting object to plot spectral data.)

Can plot a large number of data values

Each plotting object can be used to plot as many data values as needed (*unless the program runs out of memory*) .

(As mentioned earlier, I have successfully plotted two million data values in 141 full screen pages on a modest laptop computer with no difficulty whatsoever. When I pushed that total up to eight million data values in 563 full screen pages, the plotting process slowed down, but I was still able to display and examine the plots. The practical limit on my computer seems to be somewhere between two million and eight million data values.)

Multiple Page objects

A plotting object of type **PlotALot01** owns one or more **Page** objects that extend the **Frame** class. The plotting object can own as many **Page** objects as are necessary to plot all of the data that is fed to the plotting object.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen, with the data plotted on a **Canvas** object contained in each **Page** object. The **Page** showing the earliest data (*page 0*) is on the top of the stack and the **Page** showing the latest data is on the bottom of the stack.

*(The **Page** objects on the top of the stack must be physically moved in order to see the **Page** objects further down in the stack.)*

Multiple traces on each Page object

As shown in [Figure 1](#), each **Page** object contains one or more horizontal axes on which the data is plotted. The earliest data is plotted on the axis nearest the top of the **Page** moving from left to right across the **Page**. Positive data values are plotted above the axis and negative values are plotted below the axis.

When the right end of an axis is reached, the next data value is plotted on the left end of the axis immediately below it (*sometimes referred to as wrap around*). When the right end of the last axis on the **Page** is reached, a new **Page** object is automatically created and the next data value is plotted at the left end of the top axis on the new **Page** object.

Two overloaded constructors

There are two overloaded versions of the constructor for the **PlotALot01** class. One overloaded version accepts several incoming parameters allowing the user to control various aspects of the plotting format. (*An example of the use of this constructor is shown in [Figure 5](#).*) A second overloaded version accepts a title string only and sets all of the plotting parameters to default values. (*An example of the use of this constructor is shown in [Figure 1](#).*)

(You can easily modify the default values and recompile the class if you prefer different default values.)

Constructor parameters

The parameters for the version of the constructor that accepts plotting parameters are:

- String title: Title for the Frame object. This title is concatenated with the page number and the result appears in the banner at the top of the Page as shown in [Figure 1](#).
- int frameWidth: The Frame width in pixels.
- int frameHeight: The Frame height in pixels.

- int traceSpacing: Distance between trace axes in pixels.
- int sampSpace: Number of pixels dedicated to each data sample in pixels per sample. *(Must be 1 or greater.)*
- int ovalWidth: Width of an oval that is used to mark the sample value on the plot. *(See [Figure 5](#) for a good example of the ovals. Set the oval width and height parameters to zero to eliminate the ovals altogether.)*
- int ovalHeight: Height of an oval that is used to mark the sample value on the plot.

Two plotting objects for test purposes

For self-test purposes, the **main** method instantiates and feeds two independent plotting objects. Plotting parameters are specified for the first plotting object and the stack of pages for this plotting object is located 401 pixels to the right of the upper left corner of the screen. The output produced by this plotting object is shown in [Figure 5](#) below. *(The two pages in the screen shot in [Figure 5](#) were manually relocated and positioned for reasons that I will explain later.)*

Figure 5. Self-test output for PlotALot01.

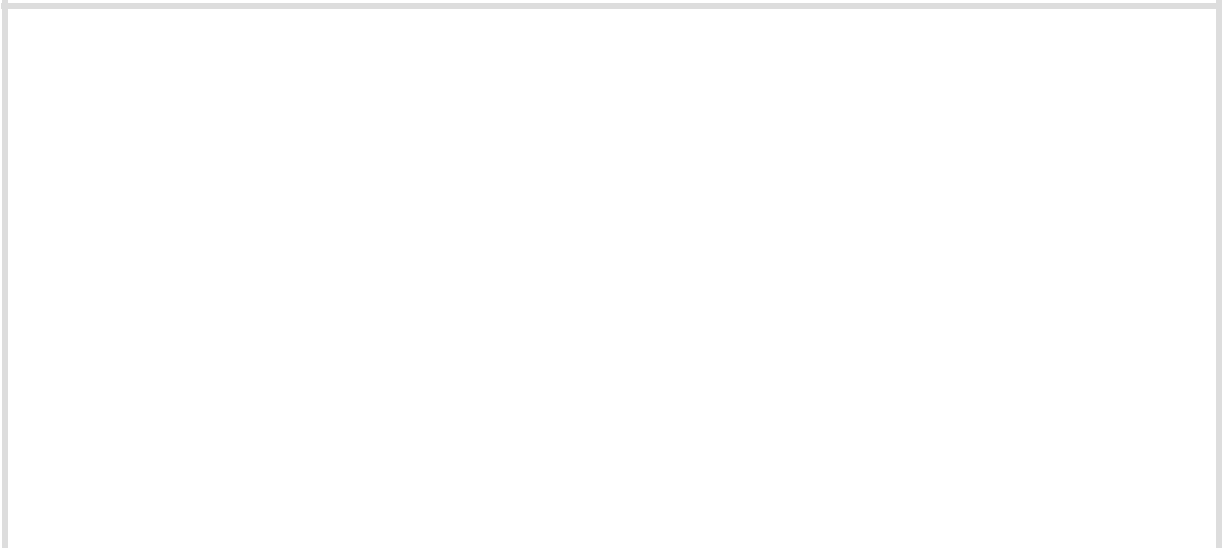
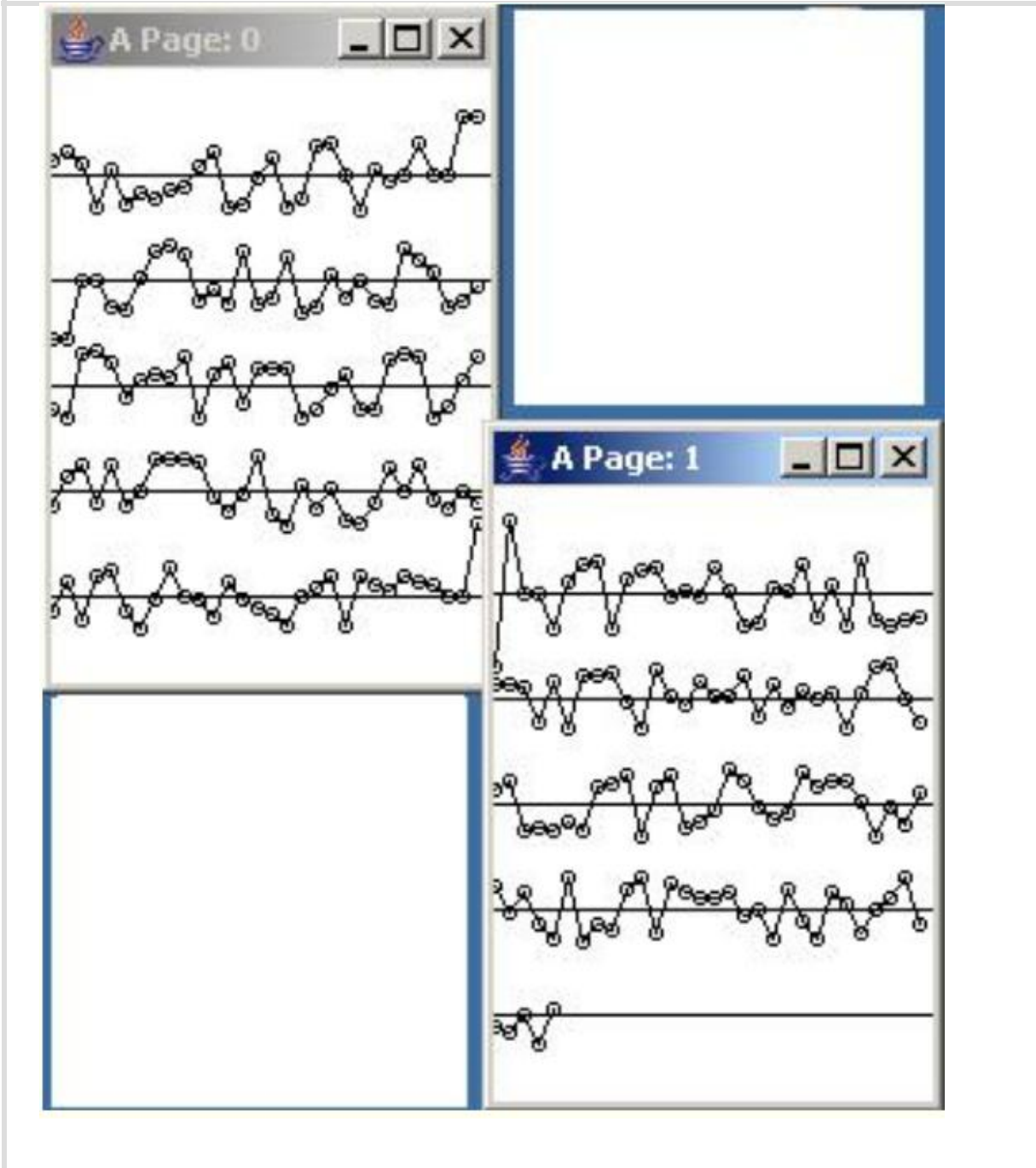


Figure 5. Self-test output for PlotALot01.



Default plotting parameters are used for the second plotting object and the stack of pages is located in the default location at the upper left corner of the screen. The output produced by this plotting object was shown earlier in [Figure 1](#).

The data to be plotted

Most of the data that is fed to each plotting object is white random noise produced by a random noise generator. However, fifteen of the data values fed to the first plotting object are not random.

Transition from trace to trace on the same page

Eight of the data values for the first plotting object are set to 0,0,20,20,-20,-20,0,0. The result can be seen at the end of the first trace and the beginning of the second trace in Page 0 in [Figure 5](#). Note that the last four plotted points for the first trace have values of 0,0,20, and 20. Then note that the first four plotted points on the second trace have values of -20, -20, 0, and 0. This confirms the proper transition from one trace to the next on the same page with no loss of data values in the transition.

Transition from page to page

Seven of the values for the first plotting object are set to values of 0,0,25,-25,25,0,0. The result can be seen at the end of the last trace on Page 0 and the beginning of the first trace on Page 1 in [Figure 5](#). Note that the last three plotted points in the last trace on Page 0 have values of 0, 0, and 25. Then note that the first four plotted points in the first trace on Page 1 have values of -25, 25, 0, and 0. This confirms the proper transition from one page to the next with no loss of data in the transition.

(The two pages in [Figure 5](#) were manually arranged as shown before capturing the screen shot to emphasize the transition of the data from one page to the next. The large white rectangles in [Figure 5](#) are the result of removing the background clutter in the image caused by icons on the desktop.)

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly by the plotting object. These are the correct locations for an AWT **Frame** object for Java running under WinXP. Note however that a **Frame** may have different *inset* values (*border widths*) under other operating systems, which may cause these specific locations to fail to match up for that operating system. In that case, the values will be plotted but they won't necessarily occur at the same physical locations in order to confirm the proper transition.

(They also match up properly for the Windows 7 Classic scheme, but not for the other Windows 7 themes.)

Information about plotting parameters

Information about the plotting parameters for each plotting object is displayed on the command line screen when this class is used for plotting. The values shown below result from the execution of the main method of the **PlotALot01** class for self-test purposes. One of the plotting objects instantiated by the main method is titled "A" and **the other is titled "B"** .

null

The graphic output produced for the object titled "A" is shown in [Figure 5](#). This output was based on plotting format parameters that were passed to the constructor. The graphic output produced for the object titled "B" is shown in [Figure 1](#). This output was based on default plotting parameters.

Overloaded plotData method

There are two overloaded versions of the **plotData** method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. This version requires two parameters, which are coordinate values in pixels. The first parameter specifies the horizontal coordinate of the upper left corner of the stack of pages relative to the upper left corner of the

screen. The second parameter specifies the vertical coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen.

(Specifying coordinate values of 0,0 causes the stack to be located in the upper left corner of the screen. Positive vertical coordinates progress down the screen.)

The other overloaded version of **plotData** places the stack of pages in the upper left corner of the screen by default.

A **WindowListener** for program termination

Each page has a **WindowListener** that will terminate the program if the user clicks the close button on the **Frame** *(the X-button in the upper-right corner)* .

J2SE 5.0 is required

The class was tested using J2SE 5.0 and WinXP. J2SE 5.0 is required because the class uses generics with an **ArrayList** object. *(More recently, it was re-tested using java version "1.8.0_60" under Windows 7.)*

Let's see some code!

I will present and explain this class in fragments. A complete listing of the class is provided in [Listing 35](#) near the end of the module.

Beginning of the class named PlotALot01

As mentioned earlier, this class contains a **main** method. The **main** method is provided so that the class can be run as an application for self-test purposes,

which is common practice in Java programming. The **main** method also illustrates the proper use of the class.

The beginning of the class and the beginning of the **main** method are shown in [Listing 1](#).

Listing 1. Beginning of the class named PlotALot01.

```
public class PlotALot01{
    public static void main(String[] args){
        PlotALot01 plotObjectA =
            new
PlotALot01("A",158,237,36,5,4,4);
        PlotALot01 plotObjectB = new
PlotALot01("B");
```

Instantiate two plotting objects

[Listing 1](#) instantiates two independent plotting objects. The first plotting object, referred to by **plotObjectA** is instantiated by calling the constructor that accepts plotting parameters. A description of each of the constructor parameters was provided [earlier](#). You may find it useful to compare the values shown in [Listing 1](#) with the [overall plotting parameters](#) listed earlier to confirm how they are related.

The second plotting object, referred to by **plotObjectB** is instantiated by calling the constructor that accepts only the page title as a parameter and uses default values for all of the plotting parameters. You will see those default values later in the code.

Feed the plotting object titled "A"

[Listing 2](#) contains a **for** loop that feeds 275 values to the plotting object titled "A". Most of the code in [Listing 2](#) is required to set fifteen specific values to test for proper transitions as described earlier. This code is straightforward and shouldn't require further explanation.

(I was able to determine the correct locations for these values by knowing the size of the Frame, inset values for the Frame, the space between traces, the number of pixels dedicated to each sample, etc.)

Listing 2. Feed the plotting object titled "A".

```
for(int cnt = 0;cnt < 275;cnt++){  
    if(cnt == 147){  
        plotObjectA.feedData(0);  
    }else if(cnt == 148){  
        plotObjectA.feedData(0);  
    }else if(cnt == 149){  
        plotObjectA.feedData(25);  
    }else if(cnt == 150){  
        plotObjectA.feedData(-25);  
    }else if(cnt == 151){  
        plotObjectA.feedData(25);  
    }else if(cnt == 152){  
        plotObjectA.feedData(0);  
    }else if(cnt == 153){  
        plotObjectA.feedData(0);  
    }else if(cnt == 26){  
        plotObjectA.feedData(0);  
    }
```

Listing 2. Feed the plotting object titled "A".

```
    }else if(cnt == 27){
        plotObjectA.feedData(0);
    }else if(cnt == 28){
        plotObjectA.feedData(20);
    }else if(cnt == 29){
        plotObjectA.feedData(20);
    }else if(cnt == 30){
        plotObjectA.feedData(-20);
    }else if(cnt == 31){
        plotObjectA.feedData(-20);
    }else if(cnt == 32){
        plotObjectA.feedData(0);
    }else if(cnt == 33){
        plotObjectA.feedData(0);
    }else{
        plotObjectA.feedData(
                                (Math.random() -
0.5)*25);
    }//end else
} //end for loop
```

White random noise

The final statement in [Listing 2](#) uses a random number generator to feed white random noise to the plotting object for all data values other than the fifteen data values specified in the preceding statements. You can see the random values plotted and marked by round ovals in [Figure 5](#).

Plot the data

The statement in [Listing 3](#) calls the overloaded **plotData** method to cause all of the pages belonging to the plotting object titled "A" to be stacked in a

location where the upper left corner of the stack is 401 pixels to the right of the upper left corner of the screen.

Listing 3. Plot the data.

```
plotObjectA.plotData(401, 0);
```

As described earlier, page 0 containing the earliest data fed to the plotting object is on the top of the stack. [Figure 1](#) shows the two pages belonging to this plotting object after they have been manually rearranged to make them both visible.

Feed and plot the object titled "B"

[Listing 4](#) feeds 2600 random white noise values to the object titled "B" and displays the pages in the default location in the upper left corner of the screen. [Listing 4](#) also signals the end of the main method.

Listing 4. Feed and plot the object titled "B".

Listing 4. Feed and plot the object titled "B".

```
for(int cnt = 0;cnt < 2600;cnt++){  
    plotObjectB.feedData(  
                                (Math.random() -  
0.5)*25);  
    }//end for loop  
    plotObjectB.plotData();  
  
    }//end main
```

[Listing 4](#) (plus one of the statements in [Listing 1](#)) is much more typical of the amount of code required to use this plotting class than was the case with [Listing 2](#).

(Almost all of the code in [Listing 2](#) was required to set the special data values used to test the transitions discussed earlier.)

The three steps

To recap, the three steps required to use this class for plotting nearly unlimited amounts of data are:

1. Instantiate a plotting object of the class named **PlotALot01** , as in [Listing 1](#).
2. Call the **feedData** method once for each data value that is to be plotted, as in [Listing 4](#).
3. Call the **plotData** method on the plotting object after all of the data has been fed to the plotting object, as in [Listing 3](#) or [Listing 4](#).

Some instance variables

Continuing with the class definition for the class named **PlotALot01** , [Listing 5](#) shows several instance variables that belong to a plotting object instantiated from this class.

Listing 5. Some instance variables.

```
String title;
int  frameWidth;
int  frameHeight;
int  traceSpacing;//pixels between traces
int  sampSpacing;//pixels between samples
int  ovalWidth;//width of sample marking oval
int  ovalHeight;//height of sample marking oval

int  tracesPerPage;
int  samplesPerPage;
int  pageCounter = 0;
int  sampleCounter = 0;
ArrayList <Page> pageLinks =
                                new ArrayList<Page>
();
```

The purpose of each of these instance variables is indicated by the name of the variable, and in some cases by the comments following the variable declaration. In addition, I will have more to say about some of these variables later when I discuss the code that uses them.

*(Note the use of [generics](#) in the declaration and initialization of the variable named **pageLinks** . The use of generics dictates that this class requires J2SE 5.0 or later.)*

The first overloaded constructor

As mentioned earlier, there are two overloaded versions of the constructor for this class. The overloaded version that begins in [Listing 6](#) accepts several incoming parameters allowing the user to control various aspects of the plotting format.

(A different overloaded version, which I will discuss later, accepts a title string only and sets all of the plotting parameters to default values.)

Listing 6. The first overloaded constructor.

```
PlotALot01(String title,//Frame title
            int frameWidth,//in pixels
            int frameHeight,//in pixels
            int traceSpacing,//in pixels
            int sampSpace,//in pixels per
sample
            int ovalWidth,//sample marker width
            int ovalHeight)//sample marker hite
{ //constructor code continues here
```

[Listing 6](#) shows the signature for this overloaded version of the constructor. The comments should make the code self explanatory.

Save the parameter values

With one exception, the code in [Listing 7](#) simply saves the incoming parameter values in instance variables to make those values available to other members of the class.

Listing 7. Save the parameter values.

```
this.title = title;
this.frameWidth = frameWidth;
this.frameHeight = frameHeight;
this.traceSpacing = traceSpacing;
//Convert to pixels between samples.
this.sampSpacing = sampSpace - 1;
this.ovalWidth = ovalWidth;
this.ovalHeight = ovalHeight;
```

The exception

The exception has to do with the parameter named **sampSpace** . This parameter is provided by the user in units of pixels per sample, because that seems to be the most natural way for a human to specify this plotting parameter. However, for computational purposes, it is better to have the value of the number of pixels between samples, which is one less than the number of pixels per sample. This conversion is made during the saving of this parameter in [Listing 7](#).

A temporary Page object

As you will see later, the **Page** class consists of a **Canvas** contained in an **AWT Frame** . Because an **AWT Frame** takes on the look and feel of the operating system under which the program is running, it may be constructed differently under different operating systems. Many important plotting parameters depend on the size of the **Canvas** , which depends on the values of the **insets** (*border width*) for the **Frame** for that particular operating system.

(A good exercise would be for you to convert this class to Swing using a look and feel that is independent of the operating system.)

The code in [Listing 8](#) instantiates a temporary **Page** object solely for the purpose of obtaining information about the width and the height of the **Canvas** object. This information is used later to compute a variety of other important parameter values.

Listing 8. A temporary Page object.

```
Page tempPage = new Page(title);
int canvasWidth =
tempPage.canvas.getWidth();
int canvasHeight =

tempPage.canvas.getHeight();
```

Display some information

[Listing 9](#) gets and displays information about the plotting object on the command line screen. An example of this output was shown earlier.

Listing 9. Display some information.

```
//Display information about this plotting
// object.
System.out.println("\nTitle: " + title);
System.out.println(
    "Frame width: " +
tempPage.getWidth());
System.out.println(
    "Frame height: " +
tempPage.getHeight());
System.out.println(
    "Page width: " +
canvasWidth);
System.out.println(
    "Page height: " +
canvasHeight);
System.out.println(
    "Trace spacing: " +
traceSpacing);
System.out.println(
    "Sample spacing: " + (sampSpacing +
1));
if(sampSpacing < 0){
    System.out.println("Terminating");
    System.exit(0);
} //end if
```

Terminate on negative value for sampSpacing

In addition, [Listing 9](#) tests to confirm that the number of pixels between samples is not a negative value. If it is a negative value, [Listing 9](#) terminates the program immediately after the number of pixels per sample has been displayed

Dispose of the temporary Page object

Once the width and height of the **Canvas** has been determined, the temporary **Page** object is no longer needed. [Listing 10](#) disposes of that object freeing all of the resources dedicated to the object.

Listing 10. Dispose of the temporary Page object.

```
tempPage.dispose();
```

Compute and display the remaining plotting parameters

Having determined the width and height of the **Canvas** , [Listing 11](#) computes and displays the remaining plotting parameters. The expressions used to compute these values are straightforward and shouldn't require further explanation.

Listing 11. Compute and display the remaining plotting parameters.

```
        tracesPerPage =
            (canvasHeight -
traceSpacing/2)/
traceSpacing;
        System.out.println("Traces per page: "
                            +
tracesPerPage);
        if(tracesPerPage == 0){
            System.out.println("Terminating program");
            System.exit(0);
        }//end if
        samplesPerPage = canvasWidth *
tracesPerPage/
                            (sampSpacing +
1);
        System.out.println("Samples per page: "
                            +
samplesPerPage);
```

Terminate on zero traces per page

In addition, [Listing 11](#) terminates the program if it is determined that the number of traces per page is equal to zero. The reason for this should be obvious to the reader. If termination does occur, it occurs immediately after the number of traces per page has been displayed.

Instantiate first usable Page object

[Listing 12](#) instantiates the first usable Page object. (Recall that a temporary Page object was instantiated and disposed of earlier.) This Page object will be

titled title Page: 0 as indicated in [Figure 1](#) and [Figure 5](#).

Listing 12. Instantiate first usable Page object.

```
        pageLinks.add(new Page(title));  
    }//end constructor
```

Note that a reference to this **Page** object (*and all subsequently instantiated Page objects*) is saved in an object of type **ArrayList** referred to by **pageLinks** .

[Listing 12](#) also signals the end of this constructor for the PlotALot01 class.

The other overloaded constructor

A second overloaded constructor is provided for those who don't want to have to think about plotting parameters. This constructor, which is shown in its entirety in [Listing 13](#), establishes a set of default plotting parameters.

*(In case you are unfamiliar with the use of the keyword **this** to cause one constructor to call another constructor of the same class, you can learn about that topic in my module titled [The Essence of OOP using Java, The this and super Keywords](#).)*

Listing 13. The other overloaded constructor .

```
PlotALot01(String title){  
    this(title,400,410,50,2,2,2);  
} //end overloaded constructor
```

The default plotting parameter values

This is where the default plotting parameters are specified as having the following values:

- `frameWidth`: 400
- `frameHeight`: 410
- `traceSpacing`: 50
- `sampSpace`: 2
- `ovalWidth`: 2
- `ovalHeight`: 2

As mentioned earlier, these values were chosen mainly to be compatible with this narrow publication format. You should feel free to change the default values to a set of values that is more consistent with your needs. For example, if you plan to plot and examine very large amounts of data, you might want to consider setting the **`frameWidth`** and **`frameHeight`** to completely fill the screen on your computer. Then you can examine large amounts of data without the need to skip from one page to the next.

The `feedData` method

The **`feedData`** method must be called on the plotting object once for each data value that is to be plotted. This method is shown in its entirety in [Listing 14](#).

Listing 14. The feedData method.

```
void feedData(double val){
    if((sampleCounter) == samplesPerPage){
        pageCounter++;
        sampleCounter = 0;
        pageLinks.add(new Page(title));
    }//end if
    pageLinks.get(pageCounter).putData(
val, sampleCounter);
    sampleCounter++;
} //end feedData
```

Scaling of data to be plotted

The **feedData** method receives an incoming data value of type **double** . This is probably a good time to point out that the data must be properly scaled for plotting before it is passed to this method.

The incoming **double** value will later be cast to type **int** . As you should already know, if the double value is too large to fit in type **int** , the value resulting from the cast will be indeterminate.

In reality, however, the cast shouldn't be a problem. I'm unaware of any computer monitor whose vertical dimension is greater than a few thousand pixels. Regardless of the size of the **Page** object, a data value whose magnitude is greater than a few thousand units will be completely off the screen when plotted. Therefore, depending on the resolution of the monitor, the maximum magnitudes of the incoming data values should probably have been scaled to 1000 or less to be suitable for plotting.

Is the page full?

[Listing 14](#) first checks to see if the current page is full before attempting to plot the new data value. If the page is full, [Listing 14](#) increments the page counter, resets the sample counter, and instantiates a new **Page** object.

Save the data value for plotting later

All of the data values are stored in array objects of type **double** as they are fed to the plotting object. Later, when it is time to display the plotted version of the data, an overridden **paint** method accesses that data and produces the plot.

The **MyCanvas** class, a preview

Each **Page** object contains an object of a class named **MyCanvas**, which extends the **Canvas** class. Each **MyCanvas** object owns an array object in which the **double** data values to be plotted on that page are stored.

The **MyCanvas** class overrides the **paint** method to cause it to plot the data stored in the array whenever the overridden version of the **paint** method is called.

*(If you are familiar with graphics in Java, you will already know that the overridden **paint** method can be called for a variety of reasons, such as covering and later uncovering the page. If you are not familiar with graphics in Java, I discuss the overriding of the **paint** method in numerous earlier modules including several modules on [animation](#) in Java.)*

I will have much more to say about the class named **MyCanvas** later.

Call the `putData` method to store the data value

For now, simply be aware that the **feedData** method in [Listing 14](#) calls the **putData** method on the current **Page** object to cause the data value to be stored in the array object belonging to the corresponding **MyCanvas** object. The current value of the sample counter is also passed to the **putData** method to specify the array element into which the data value is to be stored.

Finally, the **feedData** method increments the sample counter and returns to await being called to receive the next data sample.

The **plotData** method

The **plotData** method must be called once when all of the data has been fed to the plotting object by way of the **feedData** method. The purpose of the **plotData** method is to rearrange the **Page** objects in a stack on the screen with page 0 (*containing the earliest data*) on the top of the stack.

Having rearranged the **Page** objects, the **plotData** method causes the object on the top of the stack to become visible. This, in turn, causes its overridden **paint** method belonging to that object to be called, thus causing the data to be plotted as shown in [Figure 1](#) and [Figure 5](#).

Two overloaded versions

There are two overloaded versions of the **plotData** method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. The other version places the stack in the upper left corner of the screen by default.

Specify the location of the stack

The version of the **plotData** method that allows the user to specify the location begins in [Listing 15](#). This version receives two incoming parameters. These parameters specify the coordinates of the upper left corner of the stack of **Page** objects relative to the upper left corner of the screen. The first parameter specifies the horizontal coordinate and the second parameter

specifies the vertical coordinate, with positive vertical values going down the screen.

(Specifying both coordinate values as 0 will cause the stack to appear in the upper left corner of the screen.)

Listing 15. Beginning of the plotData method.

```
void plotData(int xCoor,int yCoor){
    Page lastPage =
        pageLinks.get(pageLinks.size() -
1);
    while(!lastPage.isVisible()){
        //Loop until last page becomes visible
    }//end while loop
```

Make certain that the last page is visible

As you will see later, each of the pages are displayed on the screen as they are produced. It is possible that this method could be called before the operating system has completed the process of making the last page visible. The **plotData** method uses a **while** loop to delay until the last page has become visible on the screen.

At this point, the pages appear on the screen with the last page on the top of the stack. This order needs to be reversed to cause the first page to be on the top of the stack.

Make all pages invisible

The reversal of the order of the pages in the stack is accomplished by first making every page invisible, and then making them visible again in reverse order.

The code in [Listing 16](#) iterates on the **ArrayList** object containing references to all of the pages. The reference to each **Page** object is obtained from the list, and its visible property value is set to false.

Listing 16. Make all pages invisible.

```
Page tempPage = null;
for(int cnt = 0; cnt < (pageLinks.size());
    cnt++)
{
    tempPage = pageLinks.get(cnt);
    tempPage.setVisible(false);
} //end for loop
```

Make the pages visible in reverse order

The code in [Listing 17](#) iterates on the **ArrayList** object again, accessing the references to the **Page** objects in reverse order and setting the value of the visible property for each **Page** object to true. This results in page 0 (*the page with the earliest data*) being on the top of the stack.

Listing 17. Make the pages visible in reverse order.

```
        for(int cnt = pageLinks.size() - 1;cnt >= 0;
                                cnt--)
    {
        tempPage = pageLinks.get(cnt);
        tempPage.setLocation(xCoor,yCoor);
        tempPage.setVisible(true);
    }//end for loop

} //end plotData(int xCoor,int yCoor)
```

Set the location of the stack

In addition, the code in [Listing 17](#) sets the location property of each **Page** object to the coordinate values received as incoming parameters to the **plotData** method. This causes the stack of **Page** objects to appear in the specified location on the screen.

The other overloaded version of the plotData method

The other overloaded version of the **plotData** method is shown in its entirety in [Listing 18](#).

Listing 18. The other overloaded version of the plotData method.

Listing 18. The other overloaded version of the plotData method.

```
void plotData(){  
    plotData(0,0); //call overloaded version  
} //end plotData()
```

This version receives no incoming parameters. The body of the method simply calls the first overloaded version discussed above, passing coordinate values as 0,0 as parameters. As explained earlier, this causes the stack of **Page** objects to appear in the upper left corner of the screen.

The Page class

The *Page* class is a [member](#) class of the class named **PlotALot01** . As such, methods belonging to objects of the **Page** class have direct access to all of the other members of the enclosing **PlotALot01** object, including instance variables belonging to the **PlotALot01** object.

(If you are unfamiliar with member classes, see the module titled [The Essence of OOP using Java, Member Classes](#).)

Potentially many **Page** objects...

A **PlotALot01** object may own as many **Page** objects as are required to plot all of the data values that are fed to it.

*(The reference to each **Page** object is stored in an **ArrayList** object belonging to the **PlotALot01** object.)*

The **Page** class, which extends the **Frame** class begins, in [Listing 19](#). The constructor for the **Page** class also begins in [Listing 19](#).

Listing 19. Beginning of the class named Page.

```
class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
        canvas = new MyCanvas();
        add(canvas);
        setSize(frameWidth,frameHeight);
        setTitle(title + " Page: " + pageCounter);
        setVisible(true);
    }
}
```

The **Page** class begins by declaring two instance variables. The instance variable named **canvas** will hold a reference to an object of the **MyCanvas** class upon which the data will actually be plotted.

The other instance variable will hold the value of a sample counter.

The constructor for the Page class

The constructor that begins in [Listing 19](#) instantiates an object of a member class named **MyCanvas** and stores its reference in the reference variable named **canvas** . Then it adds the **MyCanvas** object to the default center location of the **Page (Frame)** .

Following this, the constructor accesses the variables named **frameWidth** and **frameHeight** belonging to the enclosing **PlotALot01** object and uses those values to set the size of the **Page** .

Then the constructor accesses the **title** and **pageCounter** variables belonging to the enclosing **PlotALot01** object and uses those values to set the title for the **Page** object.

Finally, the code in [Listing 19](#) causes the **Page** object to become visible on the screen.

An anonymous terminator for the Page class

Still inside the constructor, [Listing 20](#) instantiates an object of an *anonymous inner class* to terminate the program when the user clicks on the close button on the Page (*the X-button in the upper-right corner*) .

(In case you are unfamiliar with anonymous inner classes, see my module titled [The Essence of OOP using Java, Anonymous Classes](#) .)

Listing 20. An anonymous terminator for the Page class.

Listing 20. An anonymous terminator for the Page class.

```
addWindowListener(  
    new WindowAdapter(){  
        public void windowClosing(  
                                WindowEvent e)  
    {  
        System.exit(0); //terminate program  
    } //end windowClosing()  
} //end WindowAdapter  
); //end addWindowListener  
} //end constructor
```

[Listing 20](#) also signals the end of the constructor for the **Page** class.

The putData method of the Page class

This method, which is shown in its entirety in [Listing 21](#), receives a sample value of type **double** and also receives the sample counter associated with that data value. It uses the value of the sample counter to store the data value in an array object belonging to the **MyCanvas** object.

Listing 21. The putData method of the Page class.

Listing 21. The putData method of the Page class.

```
void putData(double sampleValue,  
             int sampleCounter){  
    canvas.data[sampleCounter] = sampleValue;  
    this.sampleCounter = sampleCounter;  
} //end putData
```

In addition, the **putData** method saves the sample counter value in an instance variable to make it available to the overridden **paint** method later. This value is needed by the **paint** method so it will know how many samples to plot on the final page which probably won't be full.

The MyCanvas class

The MyCanvas class, which begins in [Listing 22](#), is a member class of the Page class. As such, methods belonging to an object of the MyCanvas class have direct access to the other members of the enclosing Page object, including instance variables of the Page object. In addition, methods belonging to an object of the MyCanvas class have direct access to the other members, including instance variables, of the enclosing PlotALot01 object.

Listing 22. Beginning of the MyCanvas class.

Listing 22. Beginning of the MyCanvas class.

```
class MyCanvas extends Canvas{
    double [] data =
                    new
double[samplesPerPage];
```

This class definition begins by creating a new array object of type **double** that will be used to store the data values belonging to the page. The size of the array is specified by the value of **samplesPerPage** .

The overridden paint method

The overridden **paint** method of the **MyCanvas** class begins in [Listing 23](#).

Listing 23. Beginning of the overridden paint method.

```
public void paint(Graphics g){
    for(int cnt = 0;cnt < tracesPerPage;
                                   cnt++)
    {
        g.drawLine(0,
                    (cnt+1)*traceSpacing,
                    this.getWidth(),
                    (cnt+1)*traceSpacing);
    }//end for loop
```

Draw the horizontal axes

The code in [Listing 23](#) draws a set of horizontal axes on the **MyCanvas** object, one for each trace that will be plotted on the object. These horizontal axes are shown in [Figure 1](#) and [Figure 5](#).

Plot the points

[Listing 24](#) shows the beginning of the code that is used to plot the data values stored in the array that was created in [Listing 22](#).

Listing 24. Beginning of code to plot the points.

```
        if(sampleCounter > 0){
            for(int cnt = 0;cnt <= sampleCounter;
                                   cnt++)
        {
            //Compute a vertical offset
            int yOffset =
                (1 + cnt*(sampSpacing + 1)/
this.getWidth())*traceSpacing;
```

[Listing 24](#) begins by testing the value of the sample counter to make certain that there are some points to be plotted. If so, it enters a **for** loop to plot each data value stored in the array. Because it uses the value of the sample counter to terminate the **for** loop, only those data values that have been stored in the array object will be plotted, even if the array object isn't full.

A vertical offset

The data values in the array are to be plotted on one or more horizontal axes on the page. Therefore, it is necessary to first determine where on the page each data value is to be plotted. The code in [Listing 24](#) uses various pieces of information to determine the vertical location of the axis against which each data value will be plotted.

Draw an oval

The code in [Listing 25](#) draws an oval centered on the sample value to mark the sample on the plot. It is best if the dimensions of the oval are evenly divisible by 2 for centering purposes. Otherwise, the ovals may appear to be a little off center.

Listing 25. Draw an oval.

```
                g.drawOval(cnt*(sampSpacing + 1)%
                        this.getWidth() -
ovalWidth/2,
                yOffset - (int)data[cnt] -
ovalHeight/2,
                        ovalWidth,
                        ovalHeight);
```

Is positive vertical up or down?

Normally vertical coordinates increase going down the screen in Java graphics. However, this isn't what most of us are accustomed to seeing when

we plot data. We prefer to see increasing vertical coordinates going up the page. The code in [Listing 25](#) reverses the sign on the data values to cause positive data values to be plotted above the axis and negative data values to be plotted below the axis. Increasing values go up. Decreasing values go down.

Connect the points with straight lines

The code in [Listing 26](#) connects the sample values with straight lines. Care is taken to avoid drawing a line from the last sample value on one trace to the first sample value on the next trace. That would really be ugly.

Listing 26. Connect the points with straight lines.

Listing 26. Connect the points with straight lines.

```
        if(cnt*(sampSpacing + 1)%
            this.getWidth()
    >=
            sampSpacing + 1)
    {
        g.drawLine(
            (cnt - 1)*(sampSpacing + 1)%
this.getWidth(),
            yOffset - (int)data[cnt-1],
            cnt*(sampSpacing + 1)%
this.getWidth(),
            yOffset - (int)data[cnt]);
    } //end if
    } //end for loop
    } //end if for sampleCounter > 0
    } //end overridden paint method
    } //end inner class MyCanvas
    } //end inner class Page
} //end class PlotALot01
```

End of the PlotALot01 class

[Listing 26](#) also signals the end of the overridden **paint** method, the end of the **MyCanvas** class, the end of the **Page** class, and the end of the **PlotALot01** class. In short, [Listing 26](#) signals the end of the class under discussion.

The class named PlotALot02

Much of the code in this class is very similar to the code in the class named **PlotALot01** . Therefore, the discussion of this class will be much briefer than the earlier discussion of the class named **PlotALot01** .

Designed for two-channel data

This class is an update to the class named **PlotALot01** . This class is designed to plot large amounts of data for two channels on the same axes as shown in [Figure 2](#) . One set of data is plotted using the color black. The other set of data is plotted using the color red.

As is the case for the class named **PlotALot01** , this class provides a **main** method so that the class can be run as an application in self-test mode.

Three steps to use the class

As before, there are three steps involved in the use of this class for plotting data:

1. Instantiate a plotting object of type **PlotALot02** .
2. Feed pairs of data values to the plotting object by calling the **feedData** method once for each pair of data values. The first value in the pair will be plotted in black. The second value in the pair will be plotted in red.
3. Call the **plotData** method on the plotting object after all of the data has been fed to the object. This causes all of the data to be plotted. It also causes the pages to be rearranged placing page 0 on the top of the stack.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen. Each **Page** object contains one or more horizontal axes on which the data is plotted as shown in [Figure 2](#) . The two data sets are superimposed on the same axes with the data from one data set being plotted in black and the data from the other data set being plotted in red.

Testing with the main method

For test purposes, the **main** method instantiates a single plotting object and feeds two data sets to that plotting object. As before, the data that is fed to the plotting object is white random noise. One of the data sets is the sequence of values obtained from a random number generator. The other data set is the same as the first except that the sign of each data values is reversed.

Some data is not random

Also as before, and for the same reason, fifteen of the data values for each data set are not random. The non-random data values are the same as in the **main** method for the class named **PlotALot01** . [Figure 2](#) illustrates how these fifteen specific values are used to confirm the proper transition from the end of one trace to the beginning of the next trace, and also to confirm the proper transition from the end of one page to the beginning of the next page.

The class named PlotALot02 and the main method

As before, I will discuss this class in fragments. A complete listing of the class is provided in [Listing 36](#) near the end of the module. However, because much of the code in this class is very similar to code that I explained for the class named **PlotALot01** , this discussion of the code will be much briefer. I will highlight those aspects of this code that are different from the code in **PlotALot01** .

The beginning of the class and an abbreviated version of the **main** method is provided in [Listing 27](#) . Much of the code has been deleted from [Listing 27](#) for brevity.

Listing 27. The class named PlotALot02 and the main method.

Listing 27. The class named PlotALot02 and the main method.

```
public class PlotALot02{
    public static void main(String[] args){
        PlotALot02 plotObjectA =
            new
PlotALot02("A",158,237,36,5,4,4);

        for(int cnt = 0;cnt < 275;cnt++){
            double valBlack = (Math.random() -
0.5)*25;
            double valRed = -valBlack;
            //Feed pairs of values to the plotting
            // object by calling the feedData method
            // once for each pair of data values.
            if(cnt == 147){
                plotObjectA.feedData(0,0);

                //...code deleted for brevity

            }else{
                plotObjectA.feedData(valBlack,valRed);
            }//end else
        }//end for loop
        //Cause the data to be plotted in the
default
        // screen location.
        plotObjectA.plotData();
    }//end main
}
```

Two data values are required

The important thing to note in [Listing 27](#) is that two data values must be passed to the **feedData** method each time it is called. This consists of one data value from each channel of data being plotted.

The feedData method

The modified **feedData** method is shown in its entirety in [Listing 28](#).

Listing 28. The feedData method.

```
void feedData(double valBlack,double valRed){
    if((sampleCounter) == samplesPerPage){
        //if the page is full, increment the page
        // counter, create a new empty page, and
        // reset the sample counter.
        pageCounter++;
        sampleCounter = 0;
        pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the MyCanvas
    // object to be used later to paint the
    // screen. Then increment the sample
    // counter. The sample values pass through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
valBlack,valRed,sampleCounter);
    sampleCounter++;
}//end feedData
```

The most significant things to note about the modified version of the **feedData** method are:

- The method receives two incoming data values as parameters instead of just one.
- The method passes the two data values, along with the sample counter value to the **putData** method of the **PlotALot02** class. Thus, the **putData** method has also been modified to require two incoming data values.

The putData method

The modified putData method is shown in its entirety in [Listing 29](#). This modified version of the method receives a pair of data values and stores each of the data values in a different array object belonging to the MyCanvas object.

Listing 29. The putData method.

```
void putData(double valBlack,double valRed,
              int sampleCounter)
{
    canvas.blackData[sampleCounter] =
valBlack;
    canvas.redData[sampleCounter] = valRed;
    this.sampleCounter = sampleCounter;
} //end putData
```

The MyCanvas class

The modified version of the **MyCanvas** class begins in [Listing 30](#).

Listing 30. Beginning of the MyCanvas class.

```
class MyCanvas extends Canvas{
    double [] blackData =
        new
double[samplesPerPage];
    double [] redData =
        new
double[samplesPerPage];
```

The class begins by creating two different array objects in which incoming data is stored instead of just one array object. These are the objects that are populated by the **putData** method in [Listing 29](#).

The overridden paint method

The modified version of the overridden paint method begins in [Listing 31](#). Most of the code was deleted for brevity from [Listing 31](#) because it is very similar to the code in the overridden paint method in the class named PlotALot01.

Listing 31. Beginning of the overridden paint method.

Listing 31. Beginning of the overridden paint method.

```
public void paint(Graphics g){
    //Draw horizontal axes
    //... code deleted for brevity

    //Plot the points.
    if(sampleCounter > 0){
        for(int cnt = 0;cnt <= sampleCounter;
                                           cnt++)
        {
            //Compute a vertical offset.
            //...code deleted for brevity

            //Begin by plotting the values from
            // the blackData array object.
            //Draw an oval.
            g.setColor(Color.BLACK);
            //...code deleted for brevity

            //Connect the sample values with
            // straight lines.
            //...code deleted for brevity
        }
    }
}
```

Setting the drawing color

The most significant thing in [Listing 31](#) is the call to the **setColor** method of the Graphics class to set the drawing color to black. Otherwise, the code is essentially the same as the code in the overridden **paint** method in **PlotALot01** . The code in [Listing 31](#) draws the black traces shown in [Figure 2](#) .

New code in the overridden paint method

The overridden version of the paint **method** continues in [Listing 32](#). The code in [Listing 32](#) is essentially all new code that was created to plot the second data set in red. However, the only real difference between this code and code that I explained earlier with respect to the class named **PlotALot01** is:

- The drawing color has been set to red instead of the default color of black.
- The data being plotted is the second set of data. This data is stored in the array object referred to by **redData** .

Otherwise, this code is essentially the same as the code that was used to plot the single data set in the overridden paint method in the class named **PlotALot01** .

Listing 32. New code in the overridden paint method.

```
        //Now plot the data stored in the
        // redData array object.
        g.setColor(Color.RED);
        //Draw the ovals as described above.
        g.drawOval(cnt*(sampSpacing + 1)%
                    this.getWidth() -
ovalWidth/2,
                    yOffset - (int)redData[cnt]
                    -
ovalHeight/2,
                    ovalWidth,
                    ovalHeight);

        //Connect the sample values with
```

Listing 32. New code in the overridden paint method.

```
        // straight lines as described
above.
        if(cnt*(sampSpacing + 1)%
                                this.getWidth()
    >=
                                sampSpacing + 1)
    {
        g.drawLine(
            (cnt - 1)*(sampSpacing + 1)%
this.getWidth(),
            yOffset - (int)redData[cnt-1],
            cnt*(sampSpacing + 1)%
this.getWidth(),
            yOffset - (int)redData[cnt]);

        }//end if
    }//end for loop
    }//end if for sampleCounter > 0
    }//end overridden paint method
    }//end inner class MyCanvas
    }//end inner class Page
} //end class PlotALot02
```

The code in [Listing 32](#) draws the red traces shown in [Figure 2](#).

End of class PlotALot02

[Listing 32](#) signals the end of the overridden **paint** method, the **MyCanvas** class, the **Page** class, and the **PlotALot02** class.

The class named **PlotALot03**

I will discuss the class named **PlotALot03** in fragments. A complete listing of the class is provided in [Listing 37](#) near the end of the module.

Much of the code in the class named **PlotALot03** is very similar to the code in **PlotALot02** . Therefore, this discussion will be brief, simply highlighting the differences between the two classes.

Two-channel data on alternating axes

This class is an update to the class named **PlotALot02** . This class is designed to plot large amounts of data for two channels on alternating horizontal axes. One set of data is plotted using the color black. The other set of data is plotted using the color red.

Three steps for using the class

As before, there are three steps involved in the use of this class for plotting two-channel data:

1. Instantiate a plotting object of type **PlotALot03** .
2. Feed pairs of data values to the plotting object by calling the **feedData** method once for each pair of data values. The first value in the pair will be plotted in black on one axis. The second value in the pair will be plotted in red on an axis below that one.
3. Call the **plotData** method on the plotting object when all of the data has been fed to the object. This causes all of the data to be plotted and also causes the **Page** objects to be rearranged so that page 0 is on the top of the stack.

A stack of **Page** objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen, with the data plotted on a **Canvas** object contained in the **Page**

object.

Each **Page** object contains two or more horizontal axes on which the data is plotted. The class will terminate if the number of axes on the page is an odd number.

Alternating axes

The two data sets are plotted on alternating axes as shown in [Figure 3](#) with the data from one data set being plotted in black on one axis and the data from the other data set being plotted in red on the axis below that axis.

The earliest data is plotted on the pair of axes nearest the top of the page moving from left to right across the page. Positive data values are plotted above the axis and negative values are plotted below the axis.

When the right end of an axis is reached, the next data value is plotted on the left end of the second axis below it skipping one axis in the process. When the right end of the last pair of axes on the page is reached, a new **Page** object is created and the next pair of data values are plotted at the left end of the top pair of axes on that new page.

Testing with the main method

For self-test purposes, the **main** method instantiates a single plotting object and feeds two data sets to that plotting object. The data that is fed to the plotting object is white random noise. One of the data sets is the sequence of values obtained from a random number generator. The other data set is the same as the first. Thus, the pairs of black and red data sets that are plotted should have the same shape making it easy to confirm that the process of plotting the two data sets is behaving the same in both cases.

Some data is not random

Fifteen of the data values for each data set are not random for the same reasons discussed earlier. [Figure 3](#) shows how these specific values confirm proper transition from one trace to the next on the same page and confirm the proper transition from one page to the next.

Modified constructor code

The first code that I will highlight as being different from the code in the class named **PlotALot02** is shown in [Listing 33](#). This code appears in the modified constructor for the **PlotALot03** class.

Listing 33. Modified constructor code.

```
        if((tracesPerPage == 0) ||  
            (tracesPerPage%2 != 0) )  
    {  
        System.out.println("Terminating program");  
        System.exit(0);  
    }//end if  
  
        samplesPerPage = canvasWidth *  
tracesPerPage/  
                                (sampSpacing +  
1)/2;
```

The **if** statement in [Listing 33](#) confirms that the number of traces per page is evenly divisible by two. If not, the program terminates.

The last statement in [Listing 33](#) computes the value of **samplesPerPage** taking into account that only half as many samples from each data set can be

plotted on a page as is the case when the plots of the two data sets are superimposed on the same axes in the class named **PlotALot02** .

The overridden paint method

Additional code that I will highlight as being different is in the overridden **paint** method of the **MyCanvas** class. This code is shown in [Listing 34](#).

Listing 34. The overridden paint method.

```
public void paint(Graphics g){
    //Draw horizontal axes
    //...code deleted for brevity

    //Plot the points
    if(sampleCounter > 0){
        for(int cnt = 0;cnt <= sampleCounter;
                                           cnt++)
        {

            //Plot values from the blackData
            // array object.
            g.setColor(Color.BLACK);

            //Compute a vertical offset to
locate
            // the black data on the odd
numbered
            // axes on the page.
            int yOffset =
                ((1 + cnt*(sampSpacing + 1)/
```

Listing 34. The overridden paint method.

```
        this.getWidth()))*2*traceSpacing)
        -
traceSpacing;

        //Draw an oval
        //...code deleted for brevity
        //Connect the sample values with
        // straight lines.
        //...code deleted for brevity

        //Plot the data stored in the
        // redData array object.
        g.setColor(Color.RED);
        //Compute a vertical offset to
locate
        // the red data on the even numbered
        // axes on the page.
        yOffset = (1 + cnt*(sampSpacing +
1)/
this.getWidth()))*2*traceSpacing;

        //Draw the ovals
        //...code deleted for brevity
        //Connect the sample values with
        // straight lines
        //...code deleted for brevity

        }//end for loop
    }//end if for sampleCounter > 0
}//end overridden paint method
}//end inner class MyCanvas
}//end inner class Page
}//end class PlotALot02
```


Some code was deleted for brevity

Most of the code in the overridden **paint** method is the same as the code that I discussed earlier and was deleted from [Listing 34](#) for brevity.

The code that is different is the code that computes the vertical offset values to locate the black data on the odd numbered axes and to locate the red data on the even numbered axes as shown in [Figure 3](#). I will let you work through the expressions in [Listing 34](#) on your own and convince yourself that the code is correct.

End of class **PlotALot03**

[Listing 34](#) signals the end of the overridden **paint** method, the **MyCanvas** class, the **Page** class, and the **PlotALot03** class.

The class named **PlotALot04**

This class is an update to the class named **PlotALot03**. This class is designed to plot large amounts of three-channel data on separate horizontal axes. One set of data is plotted using the color black. The second set of data is plotted using the color red. The third set of data is plotted using the color blue.

The class provides a **main** method so that the class can be run as an application to test itself.

Three steps for using the class

There are three steps involved in the use of this class for plotting data:

- Instantiate a plotting object of type **PlotALot04**.
- Feed triplets of data values to the plotting object by calling the **feedData** method once for each triplet of data values. The first value in the triplet will be plotted in black on one axis. The second value in the triplet will

be plotted in red on an axis below that axis. The third value in the triplet will be plotted in blue on an axis below that one.

- Call the **plotData** method on the plotting object when all of the data has been fed to the object.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen, with the data plotted on a **Canvas** object contained in the **Page** object. The page showing the earliest data is on the top of the stack and the page showing the latest data is on the bottom of the stack.

Each **Page** object contains three or more horizontal axes on which the data is plotted. The class will terminate if the number of axes on the page is not evenly divisible by 3.

The three data sets are plotted on separate axes as shown in [Figure 4](#) with the data from one data set being plotted in black on one axis, the data from the second data set being plotted in red on the axis below that axis, and the data from the third data set being plotted in blue on the axis below that axis.

Testing with the main method

For test purposes, the **main** method instantiates a single plotting object and feeds three data sets to that plotting object producing the graphic output shown in [Figure 4](#).

Won't discuss the code

The code in this class is so similar to the code in the class named PlotALot03 that I'm not going to discuss the code. You will find a complete listing of the class in [Listing 38](#) near the end of the module.

Run the programs

I encourage you to copy, compile, and run the programs that you will find in [Listing 35](#) through [Listing 38](#) below.

Modify the programs and experiment with them in order to learn as much as you can about the use of Java for plotting large quantities of data. For example, you might want to modify the default plotting parameters to a different set of plotting parameters that are more to your liking. One possibility is to cause the default Page size to fill the entire screen on your computer.

Another good exercise would be for you to convert this class to Swing using a look and feel that is independent of the operating system.

Summary

In this module, I presented and explained four self-testing classes for plotting large quantities of data. One class plots a nearly unlimited amount of single-channel data using multiple traces on multiple pages.

(I have successfully plotted two million data values in 141 full screen pages on a modest laptop computer with no difficulty whatsoever. When I pushed that total up to eight million data values in 563 full screen pages, the plotting process slowed down, but I was still able to display and examine the plots. The practical limit on my computer seems to be somewhere between two million and eight million data values.)

A second class plots a large quantity of two-channel data superimposing the two data sets on the same axes with the plot of one data set being colored black and the plot of the other data set being colored red.

A third class also plots a large quantity of two-channel data, but with this class, the two sets of data are plotted on alternating horizontal axes. Again, one set of data is colored black and the other set is colored red.

A fourth class plots a large quantity of three-channel data on separate axes. In this case, one set is colored black, the second set is colored red, and the third set is colored blue.

Complete program listings

Complete listings of the four programs that I explained in this module are provided in [Listing 35](#) through [Listing 38](#) below.

Listing 35. PlotALot01.java.

```
/*File PlotALot01.java
Copyright 2005, R.G.Baldwin
This program is designed to plot large amounts of
time-series data for a single channel.  See
PlotALot02.java for a two-channel program.
```

Note that by carefully adjusting the plotting parameters, this program could also be used to plot large quantities of spectral data in a waterfall display.

The class provides a main method so that the class can be run as an application to test itself.

There are three steps involved in the use of this class for plotting time series data:

1. Instantiate a plotting object of type PlotALot01 using one of two overloaded constructors.
2. Feed data that is to be plotted to the

Listing 35. PlotALot01.java.

- plotting object by calling the `feedData` method once for each data value.
3. call one of two overloaded `plotData` methods on the plotting object once all of the data has been fed to the object. This causes all of the data to be plotted.

A using program can instantiate as many plotting objects as are needed to plot all of the different time series that need to be plotted. Each plotting object can be used to plot as many data values as need be plotted until the program runs out of available memory.

The plotting object of type `PlotALot01` owns one or more `Page` objects that extend the `Frame` class. The plotting object can own as many `Page` objects as are necessary to plot all of the data that is fed to that plotting object.

The program produces a graphic output consisting of a stack of `Page` objects on the screen, with the data plotted on a `Canvas` object contained by the `Page` object. The `Page` showing the earliest data is on the top of the stack and the `Page` showing the latest data is on the bottom of the stack. The `Page` objects on the top of the stack must be physically moved in order to see the `Page` objects on the bottom of the stack.

Each `Page` object contains one or more horizontal axes on which the data is plotted. The earliest data is plotted on the axis nearest the top of the `Page` moving from left to right across the axis. Positive data values are plotted above the axis and negative values are plotted below

Listing 35. PlotALot01.java.

the axis. When the right end of an axis is reached, the next data value is plotted on the left end of the axis immediately below it. When the right end of the last axis on the Page is reached, a new Page object is created and the next data value is plotted at the left end of the top axis on that Page object.

As mentioned above, there are two overloaded versions of the constructor for the PlotALot01 class. One overloaded version accepts several incoming parameters allowing the user to control various aspects of the plotting format. A second overloaded version accepts a title string only and sets all of the plotting parameters to default values. You can easily modify these default values and recompile the class if you prefer different default values.

The parameters for the version of the constructor that accepts plotting format information are:

String title: Title for the Frame object. This title is concatenated with the page number and the result appears in the banner at the top of the Frame.

int frameWidth: The Frame width in pixels.

int frameHeight: The Frame height in pixels.

int traceSpacing: Distance between trace axes in pixels.

int sampSpace: Number of pixels dedicated to each data sample in pixels per sample. Must be 1 or greater.

int ovalWidth: Width of an oval that is used to mark the sample value on the plot.

int ovalHeight: Height of an oval that is used to

Listing 35. PlotALot01.java.

mark the sample value on the plot.

For test purposes, the main method instantiates and feeds two independent plotting objects. Plotting parameters are specified for the first plotting object. Default plotting parameters are accepted for the second plotting object.

The data that is fed to each plotting object is white random noise. However, for the first plotting object, fifteen of the data values are not random. Rather, seven of the values are set to values of 0,0,25,-25,25,0,0 to confirm the proper transition from the end of one page to the beginning of the next page. In addition, eight of the values are set to 0,0,20,20,-20,-20,0,0 in order to confirm the proper transition from one trace to the next trace on the same page.

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly. Note, however that these are the correct locations for an AWT Frame object under WinXP. A Frame may have different inset values under other operating systems, which may cause these specific locations to be incorrect for that operating system. In that case, the values will be plotted but they won't confirm the proper transition.

The following information about the plotting parameters for each plotting object is displayed on the command line screen when the class is used for plotting. The values shown below result from the execution of the main method of the class for

Listing 35. PlotALot01.java.

test purposes. One of the plotting objects instantiated by the main method is titled "A" and the other is titled "B".

Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
Traces per page: 5
Samples per page: 150

Title: B
Frame width: 400
Frame height: 410
Page width: 392
Page height: 383
Trace spacing: 50
Sample spacing: 2
Traces per page: 7
Samples per page: 1372

There are two overloaded versions of the plotData method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. This version requires two parameters, which are coordinate values in pixels. The first parameter specifies the horizontal coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. The second parameter specifies the vertical coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. Specifying

Listing 35. PlotALot01.java.

coordinate values of 0,0 causes the stack to be located in the upper left corner of the screen.

The other overloaded version of plotData places the stack of pages in the upper left corner of the screen by default.

Each page has a WindowListener that will terminate the program if the user clicks the close button on the Frame.

The program was tested using J2SE 5.0 and WinXP. Requires J2SE 5.0 to support generics.

*****/

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot01{
    //This main method is provided so that the
    // class can be run as an application to test
    // itself.
    public static void main(String[] args){
        //Instantiate two independent plotting
        // objects. Control plotting parameters for
        // the first object. Accept default plotting
        // parameters for the second object.
        PlotALot01 plotObjectA =
            new PlotALot01("A",158,237,36,5,4,4);
        PlotALot01 plotObjectB = new PlotALot01("B");

        //Feed the data to the first plotting object.
        for(int cnt = 0;cnt < 275;cnt++){
            //Plot some white random noise in the first
            // object using specified plotting
```

Listing 35. PlotALot01.java.

```
// parameters. Note, that fifteen of the
// following values are not random. Seven
// values are set to 0,0,25,-25,25,0,0
// specifically to confirm the proper
// transition from the end of one page to
// the beginning of the next page. Eight
// values are set to 0,0,20,20,-20,-20,0,0
// to confirm the proper transition from
// one trace to the next trace on the same
// page. Note that these are the correct
// values for an AWT Frame object under
// WinXP. However, a Frame may have
// different inset values on other
// operating systems, which may cause these
// specific values to be incorrect.
if(cnt == 147){
    plotObjectA.feedData(0);
}else if(cnt == 148){
    plotObjectA.feedData(0);
}else if(cnt == 149){
    plotObjectA.feedData(25);
}else if(cnt == 150){
    plotObjectA.feedData(-25);
}else if(cnt == 151){
    plotObjectA.feedData(25);
}else if(cnt == 152){
    plotObjectA.feedData(0);
}else if(cnt == 153){
    plotObjectA.feedData(0);
}else if(cnt == 26){
    plotObjectA.feedData(0);
}else if(cnt == 27){
    plotObjectA.feedData(0);
}else if(cnt == 28){
    plotObjectA.feedData(20);
}else if(cnt == 29){
```

Listing 35. PlotALot01.java.

```
        plotObjectA.feedData(20);
    }else if(cnt == 30){
        plotObjectA.feedData(-20);
    }else if(cnt == 31){
        plotObjectA.feedData(-20);
    }else if(cnt == 32){
        plotObjectA.feedData(0);
    }else if(cnt == 33){
        plotObjectA.feedData(0);
    }else{
        plotObjectA.feedData(
                                (Math.random() - 0.5)*25);
    }
} //end else
} //end for loop
//Cause the data to be plotted.
plotObjectA.plotData(401,0);

//Plot white random noise in the second
// plotting object using default plotting
// parameters.
//Feed the data to the second plotting
// object.
for(int cnt = 0;cnt < 2600;cnt++){
    plotObjectB.feedData(
                                (Math.random() - 0.5)*25);
} //end for loop
//Cause the data to be plotted.
plotObjectB.plotData();

} //end main
//-----//

String title;
int frameWidth;
int frameHeight;
int traceSpacing; //pixels between traces
```

Listing 35. PlotALot01.java.

```
int sampSpacing;//pixels between samples
int ovalWidth;//width of sample marking oval
int ovalHeight;//height of sample marking oval

int tracesPerPage;
int samplesPerPage;
int pageCount = 0;
int sampleCounter = 0;
ArrayList <Page> pageLinks =
                                new ArrayList<Page>();

//There are two overloaded versions of the
// constructor for this class. This
// overloaded version accepts several incoming
// parameters allowing the user to control
// various aspects of the plotting format. A
// different overloaded version accepts a title
// string only and sets all of the plotting
// parameters to default values.
PlotALot01(String title,//Frame title
            int frameWidth,//in pixels
            int frameHeight,//in pixels
            int traceSpacing,//in pixels
            int sampSpace,//in pixels per sample
            int ovalWidth,//sample marker width
            int ovalHeight)//sample marker hite
{
    //constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1. Convert to
    // pixels between samples for purposes of
    // computation.
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between samples.
```

Listing 35. PlotALot01.java.

```
this.sampSpacing = sampSpace - 1;
this.ovalWidth = ovalWidth;
this.ovalHeight = ovalHeight;

//The following object is instantiated solely
// to provide information about the width and
// height of the canvas. This information is
// used to compute a variety of other
// important values.
Page tempPage = new Page(title);
int canvasWidth = tempPage.canvas.getWidth();
int canvasHeight =
    tempPage.canvas.getHeight();
//Display information about this plotting
// object.
System.out.println("\nTitle: " + title);
System.out.println(
    "Frame width: " + tempPage.getWidth());
System.out.println(
    "Frame height: " + tempPage.getHeight());
System.out.println(
    "Page width: " + canvasWidth);
System.out.println(
    "Page height: " + canvasHeight);
System.out.println(
    "Trace spacing: " + traceSpacing);
System.out.println(
    "Sample spacing: " + (sampSpacing + 1));
if(sampSpacing < 0){
    System.out.println("Terminating");
    System.exit(0);
} //end if
//Get rid of this temporary page.
tempPage.dispose();
//Now compute the remaining important values.
tracesPerPage =
```

Listing 35. PlotALot01.java.

```
                (canvasHeight - traceSpacing/2)/
                traceSpacing;
System.out.println("Traces per page: "
                + tracesPerPage);
if(tracesPerPage == 0){
    System.out.println("Terminating program");
    System.exit(0);
} //end if
samplesPerPage = canvasWidth * tracesPerPage/
                (sampSpacing + 1);
System.out.println("Samples per page: "
                + samplesPerPage);
//Now instantiate the first usable Page
// object and store its reference in the
// list.
pageLinks.add(new Page(title));
} //end constructor
//-----//

PlotALot01(String title){
    //call the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
} //end overloaded constructor
//-----//

//call this method for each point to be
// plotted.
void feedData(double val){
    if((sampleCounter) == samplesPerPage){
        //if the page is full, increment the page
        // counter, create a new empty page, and
        // reset the sample counter.
        pageCount++;
        sampleCounter = 0;
    }
}
```

Listing 35. PlotALot01.java.

```
        pageLinks.add(new Page(title));
    }//end if
    //Store the sample value in the MyCanvas
    // object to be used later to paint the
    // screen. Then increment the sample
    // counter. The sample value passes through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
                                                val,sampleCounter);

    sampleCounter++;
} //end feedData
//-----//

//There are two overloaded versions of the
// plotData method. One version allows the
// user to specify the location on the screen
// where the stack of plotted pages will
// appear. The other version places the stack
// in the upper left corner of the screen.

//call one of the overloaded versions of
// this method once when all of the data has
// been fed to the plotting object in order to
// rearrange the order of the pages with
// page 0 at the top of the stack on the
// screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen. Values of 0,0 will
// place the stack at the upper left corner of
// the screen. Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
```

Listing 35. PlotALot01.java.

```
    Page lastPage =
        pageLinks.get(pageLinks.size() - 1);
    //Delay until last page becomes visible.
    while(!lastPage.isVisible()){
        //Loop until last page becomes visible
    }//end while loop

    Page tempPage = null;
    //Make all pages invisible
    for(int cnt = 0;cnt < (pageLinks.size());
        cnt++){
        tempPage = pageLinks.get(cnt);
        tempPage.setVisible(false);
    }//end for loop

    //Now make all pages visible in reverse order
    // so that page 0 will be on top of the
    // stack on the screen.
    for(int cnt = pageLinks.size() - 1;cnt >= 0;
        cnt--){
        tempPage = pageLinks.get(cnt);
        tempPage.setLocation(xCoor,yCoor);
        tempPage.setVisible(true);
    }//end for loop

}//end plotData(int xCoor,int yCoor)
//-----//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
void plotData(){
    plotData(0,0);//call overloaded version
}//end plotData()
//-----//
```


Listing 35. PlotALot01.java.

```
//Inner class. A PlotALot01 object may
// have as many Page objects as are required
// to plot all of the data values. The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot01 object.
class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
        canvas = new MyCanvas();
        add(canvas);
        setSize(frameWidth,frameHeight);
        setTitle(title + " Page: " + pageCounter);
        setVisible(true);

        //-----//
        //Anonymous inner class to terminate the
        // program when the user clicks the close
        // button on the Frame.
        addWindowListener(
            new WindowAdapter(){
                public void windowClosing(
                    WindowEvent e){
                    System.exit(0);//terminate program
                }//end windowClosing()
            }//end WindowAdapter
        );//end addWindowListener
        //-----//
    }//end constructor
    //=====//

    //This method receives a sample value of type
    // double and stores it in an array object
    // belonging to the MyCanvas object.
```

Listing 35. PlotALot01.java.

```
void putData(double sampleValue,
             int sampleCounter){
    canvas.data[sampleCounter] = sampleValue;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
} //end putData

//=====//
//Inner class
class MyCanvas extends Canvas{
    double [] data =
        new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
        //Draw horizontal axes, one for each
        // trace.
        for(int cnt = 0; cnt < tracesPerPage;
            cnt++){
            g.drawLine(0,
                (cnt+1)*traceSpacing,
                this.getWidth(),
                (cnt+1)*traceSpacing);
        } //end for loop

        //Plot the points if there are any to be
        // plotted.
        if(sampleCounter > 0){
            for(int cnt = 0; cnt <= sampleCounter;
                cnt++){
                //Compute a vertical offset to locate
```

Listing 35. PlotALot01.java.

```
// the data on a particular trace.
int yOffset =
    (1 + cnt*(sampSpacing + 1)/
    this.getWidth())*traceSpacing;
//Draw an oval centered on the sample
// value to mark the sample. It is
// best if the dimensions of the oval
// are evenly divisible by 2 for
// centering purposes.
//Reverse the sign on sample value to
// cause positive sample values to go
// up on the screen
g.drawOval(cnt*(sampSpacing + 1)%
    this.getWidth() - ovalWidth/2,
    yOffset - (int)data[cnt]
    - ovalHeight/2,
    ovalWidth,
    ovalHeight);

//Connect the sample values with
// straight lines. Do not draw a
// line connecting the last sample in
// one trace to the first sample in
// the next trace.
if(cnt*(sampSpacing + 1)%
    this.getWidth() >=
    sampSpacing + 1){
    g.drawLine(
        (cnt - 1)*(sampSpacing + 1)%
        this.getWidth(),
        yOffset - (int)data[cnt-1],
        cnt*(sampSpacing + 1)%
        this.getWidth(),
        yOffset - (int)data[cnt]);
} //end if
} //end for loop
```

Listing 35. PlotALot01.java.

```
        }//end if for sampleCounter > 0
    }//end overridden paint method
} //end inner class MyCanvas
} //end inner class Page
} //end class PlotALot01
//=====//
```

Listing 36. PlotALot02.java.

```
/*File PlotALot02.java
Copyright 2005, R.G.Baldwin
This program is an update to the program named
PlotALot01. This program is designed to plot
large amounts of time-series data for two
channels on the same axes. One set of data is
plotted using the color black. The other set of
data is plotted using the color red. See
PlotALot01.java for a one-channel program.
```

Note that by carefully adjusting the plotting parameters, this program could also be used to plot large quantities of spectral data in a waterfall display.

The class provides a main method so that the class can be run as an application to test itself.

There are three steps involved in the use of this

Listing 36. PlotALot02.java.

class for plotting time series data:

1. Instantiate a plotting object of type PlotALot02 using one of two overloaded constructors.
2. Feed pairs of data values that are to be plotted to the plotting object by calling the feedData method once for each pair of data values. The first value in the pair will be plotted in black. The second value in the pair will be plotted in red.
3. call one of two overloaded plotData methods on the plotting object once all of the data has been fed to the object. This causes all of the data to be plotted.

A using program can instantiate as many plotting objects as are needed to plot all of the different pairs of time series that need to be plotted. Each plotting object can be used to plot as many pairs of data values as need be plotted until the program runs out of available memory.

The plotting object of type PlotALot02 owns one or more Page objects that extend the Frame class. The plotting object can own as many Page objects as are necessary to plot all of the pairs of data that are fed to that plotting object.

The program produces a graphic output consisting of a stack of Page objects on the screen, with the data plotted on a Canvas object contained by the Page object. The Page showing the earliest data is on the top of the stack and the Page showing the latest data is on the bottom of the stack. The Page objects on the top of the stack

Listing 36. PlotALot02.java.

must be physically moved in order to see the Page objects on the bottom of the stack.

Each Page object contains one or more horizontal axes on which the data is plotted. The two time series are superimposed on the same axes with the data from one time series being plotted in black and the data from the other time series being plotted in red.

The earliest data is plotted on the axis nearest the top of the Page moving from left to right across the horizontal axis. Positive data values are plotted above the axis and negative values are plotted below the axis. When the right end of an axis is reached, the next data value is plotted on the left end of the axis immediately below it. When the right end of the last axis on the Page is reached, a new Page object is created and the next data value is plotted at the left end of the top axis on that new Page object.

As mentioned above, there are two overloaded versions of the constructor for the PlotALot02 class. One overloaded version accepts several incoming parameters allowing the user to control various aspects of the plotting format. A second overloaded version accepts a title string only and sets all of the plotting parameters to default values. You can easily modify these default values and recompile the class if you prefer different default values.

The parameters for the version of the constructor that accepts plotting format information are:

Listing 36. PlotALot02.java.

String title: Title for the Frame object. This title is concatenated with the page number and the result appears in the banner at the top of the Frame.

int frameWidth: The Frame width in pixels.

int frameHeight: The Frame height in pixels.

int traceSpacing: Distance between trace axes in pixels.

int sampSpace: Number of pixels dedicated to each data sample in pixels per sample. Must be 1 or greater.

int ovalWidth: Width of an oval that is used to mark each sample value on the plot.

int ovalHeight: Height of an oval that is used to mark each sample value on the plot.

For test purposes, the main method instantiates a single plotting object and feeds two time series to that plotting object. Plotting parameters are specified for the plotting object by using the overloaded version of the constructor that accepts plotting parameters.

The data that is fed to the plotting object is white random noise. One of the time series is the sequence of values obtained from a random number generator. The other time series is the same as the first except that the sign of each data values are reversed.

Fifteen of the data values for each time series are not random. Seven of the values for the first time series are set to values of 0,0,25,-25,25,0,0. The corresponding seven values for the second time series are set to the same values

Listing 36. PlotALot02.java.

with sign reversal. This is done to confirm the proper transition from the end of one page to the beginning of the next page.

In addition, eight of the values for the first time series are set to 0,0,20,20,-20,-20,0,0. The corresponding values for the second time series are set to the same values with sign reversal. This is done in order to confirm the proper transition from one trace to the next trace on the same page.

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly. Note, however that these are the correct locations for an AWT Frame object under WinXP. A Frame may have different inset values under other operating systems, which may cause these specific locations to be incorrect for that operating system. In that case, the values will be plotted but they won't confirm the proper transition.

The following information about the plotting parameters is displayed on the command line screen when the class is used for plotting. The values shown below result from the execution of the main method of the class for test purposes.

Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36

Listing 36. PlotALot02.java.

Sample spacing: 5
Traces per page: 5
Samples per page: 150

There are two overloaded versions of the `plotData` method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. This version requires two parameters, which are coordinate values in pixels. The first parameter specifies the horizontal coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. The second parameter specifies the vertical coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. Specifying coordinate values of 0,0 causes the stack to be located in the upper left corner of the screen.

The other overloaded version of `plotData` places the stack of pages in the upper left corner of the screen by default. The main method in this class uses the second version causing the stack of pages to appear in the upper left corner of the screen by default.

Each page has a `WindowListener` that will terminate the program if the user clicks the close button on the `Frame`.

The program was tested using J2SE 5.0 and WinXP. Requires J2SE 5.0 to support generics.

```
*****/  
  
import java.awt.*;  
import java.awt.event.*;
```

Listing 36. PlotALot02.java.

```
import java.util.*;

public class PlotALot02{
    //This main method is provided so that the
    // class can be run as an application to test
    // itself.
    public static void main(String[] args){
        //Instantiate a plotting object using the
        // version of the constructor that allows for
        // controlling the plotting parameters.
        PlotALot02 plotObjectA =
            new PlotALot02("A",158,237,36,5,4,4);

        //Feed pairs of data values to the plotting
        // object.
        for(int cnt = 0;cnt < 275;cnt++){
            //Plot some white random noise Note that
            // fifteen of the values for each time
            // series are not random. See the opening
            // comments for a discussion of the reasons
            // why. Cause the values for the second
            // time series to be the negative of the
            // values for the first time series.
            double valBlack = (Math.random() - 0.5)*25;
            double valRed = -valBlack;
            //Feed pairs of values to the plotting
            // object by calling the feedData method
            // once for each pair of data values.
            if(cnt == 147){
                plotObjectA.feedData(0,0);
            }else if(cnt == 148){
                plotObjectA.feedData(0,0);
            }else if(cnt == 149){
                plotObjectA.feedData(25, -25);
            }else if(cnt == 150){
                plotObjectA.feedData(-25,25);
            }
        }
    }
}
```

Listing 36. PlotALot02.java.

```
        }else if(cnt == 151){
            plotObjectA.feedData(25, -25);
        }else if(cnt == 152){
            plotObjectA.feedData(0,0);
        }else if(cnt == 153){
            plotObjectA.feedData(0,0);
        }else if(cnt == 26){
            plotObjectA.feedData(0,0);
        }else if(cnt == 27){
            plotObjectA.feedData(0,0);
        }else if(cnt == 28){
            plotObjectA.feedData(20, -20);
        }else if(cnt == 29){
            plotObjectA.feedData(20, -20);
        }else if(cnt == 30){
            plotObjectA.feedData(-20,20);
        }else if(cnt == 31){
            plotObjectA.feedData(-20,20);
        }else if(cnt == 32){
            plotObjectA.feedData(0,0);
        }else if(cnt == 33){
            plotObjectA.feedData(0,0);
        }else{
            plotObjectA.feedData(valBlack,valRed);
        }//end else
    }//end for loop
    //Cause the data to be plotted in the default
    // screen location.
    plotObjectA.plotData();
}//end main
//-----//

String title;
int framewidth;
int frameHeight;
int traceSpacing;//pixels between traces
```

Listing 36. PlotALot02.java.

```
int sampSpacing;//pixels between samples
int ovalWidth;//width of sample marking oval
int ovalHeight;//height of sample marking oval

int tracesPerPage;
int samplesPerPage;
int pageCounter = 0;
int sampleCounter = 0;
ArrayList <Page> pageLinks =
                                new ArrayList<Page>();

//There are two overloaded versions of the
// constructor for this class. This
// overloaded version accepts several incoming
// parameters allowing the user to control
// various aspects of the plotting format. A
// different overloaded version accepts a title
// string only and sets all of the plotting
// parameters to default values.
PlotALot02(String title,//Frame title
            int frameWidth,//in pixels
            int frameHeight,//in pixels
            int traceSpacing,//in pixels
            int sampSpace,//in pixels per sample
            int ovalWidth,//sample marker width
            int ovalHeight)//sample marker hite
{
    //constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1. Convert to
    // pixels between samples for purposes of
    // computation.
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between samples.
```

Listing 36. PlotALot02.java.

```
this.sampSpacing = sampSpace - 1;
this.ovalWidth = ovalWidth;
this.ovalHeight = ovalHeight;

//The following object is instantiated solely
// to provide information about the width and
// height of the canvas. This information is
// used to compute a variety of other
// important values.
Page tempPage = new Page(title);
int canvasWidth = tempPage.canvas.getWidth();
int canvasHeight =
    tempPage.canvas.getHeight();
//Display information about this plotting
// object.
System.out.println("\nTitle: " + title);
System.out.println(
    "Frame width: " + tempPage.getWidth());
System.out.println(
    "Frame height: " + tempPage.getHeight());
System.out.println(
    "Page width: " + canvasWidth);
System.out.println(
    "Page height: " + canvasHeight);
System.out.println(
    "Trace spacing: " + traceSpacing);
System.out.println(
    "Sample spacing: " + (sampSpacing + 1));
if(sampSpacing < 0){
    System.out.println("Terminating");
    System.exit(0);
} //end if
//Get rid of this temporary page.
tempPage.dispose();
//Now compute the remaining important values.
tracesPerPage =
```

Listing 36. PlotALot02.java.

```
                (canvasHeight - traceSpacing/2)/
                traceSpacing;
System.out.println("Traces per page: "
                + tracesPerPage);
if(tracesPerPage == 0){
    System.out.println("Terminating program");
    System.exit(0);
} //end if
samplesPerPage = canvasWidth * tracesPerPage/
                (sampSpacing + 1);
System.out.println("Samples per page: "
                + samplesPerPage);
//Now instantiate the first usable Page
// object and store its reference in the
// list.
pageLinks.add(new Page(title));
} //end constructor
//-----//

PlotALot02(String title){
    //call the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
} //end overloaded constructor
//-----//

//call this method once for each pair of data
// values to be plotted.
void feedData(double valBlack,double valRed){
    if((sampleCounter) == samplesPerPage){
        //if the page is full, increment the page
        // counter, create a new empty page, and
        // reset the sample counter.
        pageCount++;
        sampleCounter = 0;
    }
}
```

Listing 36. PlotALot02.java.

```
        pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the MyCanvas
    // object to be used later to paint the
    // screen. Then increment the sample
    // counter. The sample values pass through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
        valBlack, valRed, sampleCounter);
    sampleCounter++;
} //end feedData
//-----//

//There are two overloaded versions of the
// plotData method. One version allows the
// user to specify the location on the screen
// where the stack of plotted pages will
// appear. The other version places the stack
// in the upper left corner of the screen.

//call one of the overloaded versions of
// this method once when all data has been fed
// to the plotting object in order to rearrange
// the order of the pages with page 0 at the
// top of the stack on the screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen. Values of 0,0 will
// place the stack at the upper left corner of
// the screen. Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
    Page lastPage =
```

Listing 36. PlotALot02.java.

```
        pageLinks.get(pageLinks.size() - 1);
//Delay until last page becomes visible.
while(!lastPage.isVisible()){
    //Loop until last page becomes visible
}//end while loop

Page tempPage = null;
//Make all pages invisible
for(int cnt = 0;cnt < (pageLinks.size());
    cnt++){

    tempPage = pageLinks.get(cnt);
    tempPage.setVisible(false);
}//end for loop

//Now make all pages visible in reverse order
// so that page 0 will be on top of the
// stack on the screen.
for(int cnt = pageLinks.size() - 1;cnt >= 0;
    cnt--){

    tempPage = pageLinks.get(cnt);
    tempPage.setLocation(xCoor,yCoor);
    tempPage.setVisible(true);
}//end for loop

}//end plotData(int xCoor,int yCoor)
//-----//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
void plotData(){
    plotData(0,0);//call overloaded version
}//end plotData()
//-----//

//Inner class.  A PlotALot02 object may
```


Listing 36. PlotALot02.java.

```
// have as many Page objects as are required
// to plot all of the data values. The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot02 object.
class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
        canvas = new MyCanvas();
        add(canvas);
        setSize(frameWidth,frameHeight);
        setTitle(title + " Page: " + pageCounter);
        setVisible(true);

        //-----//
        //Anonymous inner class to terminate the
        // program when the user clicks the close
        // button on the Frame.
        addWindowListener(
            new WindowAdapter(){
                public void windowClosing(
                    WindowEvent e){
                    System.exit(0);//terminate program
                }//end windowClosing()
            }//end WindowAdapter
        );//end addWindowListener
        //-----//
    }//end constructor
    //=====//

    //This method receives a pair of sample
    // values of type double and stores each of
    // them in a separate array object belonging
    // to the MyCanvas object.
```

Listing 36. PlotALot02.java.

```
void putData(double valBlack, double valRed,
             int sampleCounter){
    canvas.blackData[sampleCounter] = valBlack;
    canvas.redData[sampleCounter] = valRed;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
} //end putData

//=====//
//Inner class
class MyCanvas extends Canvas{
    double [] blackData =
        new double[samplesPerPage];
    double [] redData =
        new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
        //Draw horizontal axes, one for each
        // trace.
        for(int cnt = 0; cnt < tracesPerPage;
            cnt++){
            g.drawLine(0,
                (cnt+1)*traceSpacing,
                this.getWidth(),
                (cnt+1)*traceSpacing);
        } //end for loop

        //Plot the points if there are any to be
        // plotted.
        if(sampleCounter > 0){
```

Listing 36. PlotALot02.java.

```
for(int cnt = 0; cnt <= sampleCounter; cnt++){
    //Compute a vertical offset to locate
    // the data on a particular trace.
    int yOffset =
        (1 + cnt*(sampSpacing + 1)/
        this.getWidth())*traceSpacing;
    //Begin by plotting the values from
    // the blackData array object.
    //Draw an oval centered on the sample
    // value to mark the sample in the
    // plot. It is best if the dimensions
    // of the oval are evenly divisible
    // by 2 for centering purposes.
    //Reverse the sign of the sample
    // value to cause positive sample
    // values to be plotted above the
    // axis.
    g.setColor(Color.BLACK);
    g.drawOval(cnt*(sampSpacing + 1)%
        this.getWidth() - ovalWidth/2,
        yOffset - (int)blackData[cnt]
            - ovalHeight/2,
        ovalWidth,
        ovalHeight);

    //Connect the sample values with
    // straight lines. Do not draw a
    // line connecting the last sample in
    // one trace to the first sample in
    // the next trace.
    if(cnt*(sampSpacing + 1)%
        this.getWidth() >=
        sampSpacing + 1){
        g.drawLine(
            (cnt - 1)*(sampSpacing + 1)%
```

Listing 36. PlotALot02.java.

```

                                this.getWidth(),
                                yOffset - (int)blackData[cnt-1],
                                cnt*(sampSpacing + 1)%
                                this.getWidth(),
                                yOffset - (int)blackData[cnt]);
} //end if

//Now plot the data stored in the
// redData array object.
g.setColor(Color.RED);
//Draw the ovals as described above.
g.drawOval(cnt*(sampSpacing + 1)%
            this.getWidth() - ovalWidth/2,
            yOffset - (int)redData[cnt]
            - ovalHeight/2,

            ovalWidth,
            ovalHeight);

//Connect the sample values with
// straight lines as described above.
if(cnt*(sampSpacing + 1)%
    this.getWidth() >=
    sampSpacing + 1){
    g.drawLine(
        (cnt - 1)*(sampSpacing + 1)%
            this.getWidth(),
        yOffset - (int)redData[cnt-1],
        cnt*(sampSpacing + 1)%
            this.getWidth(),
        yOffset - (int)redData[cnt]);
} //end if
} //end for loop
} //end if for sampleCounter > 0
} //end overridden paint method
} //end inner class MyCanvas
```

Listing 36. PlotALot02.java.

```
    }//end inner class Page  
}//end class PlotALot02  
//=====//
```

Listing 37. PlotALot03.java.

```
/*File PlotALot03.java  
Copyright 2005, R.G.Baldwin  
This program is an update to the program named  
PlotALot02. This program is designed to plot  
large amounts of time-series data for two  
channels on alternating horizontal axes. One set  
of data is plotted using the color black. The  
other set of data is plotted using the color red.
```

See PlotALot02 for a class that plots two channels of data in black and red superimposed on the same axes. See PlotALot01.java for a one-channel program.

Note that by carefully adjusting the plotting parameters, this program could also be used to plot large quantities of spectral data in a waterfall display.

The class provides a main method so that the class can be run as an application to test itself.

Listing 37. PlotALot03.java.

There are three steps involved in the use of this class for plotting time series data:

1. Instantiate a plotting object of type PlotALot03 using one of two overloaded constructors.
2. Feed pairs of data values that are to be plotted to the plotting object by calling the feedData method once for each pair of data values. The first value in the pair will be plotted in black on one axis. The second value in the pair will be plotted in red on an axis below that axis.
3. call one of two overloaded plotData methods on the plotting object once all of the data has been fed to the object. This causes all of the data to be plotted.

A using program can instantiate as many plotting objects as are needed to plot all of the different pairs of time series that need to be plotted. Each plotting object can be used to plot as many pairs of data values as need be plotted until the program runs out of available memory.

The plotting object of type PlotALot03 owns one or more Page objects that extend the Frame class. The plotting object can own as many Page objects as are necessary to plot all of the pairs of data that are fed to that plotting object.

The program produces a graphic output consisting of a stack of Page objects on the screen, with the data plotted on a Canvas object contained by the Page object. The Page showing the earliest data is on the top of the stack and the Page

Listing 37. PlotALot03.java.

showing the latest data is on the bottom of the stack. The Page objects on the top of the stack must be physically moved in order to see the Page objects on the bottom of the stack.

Each Page object contains two or more horizontal axes on which the data is plotted. The program will terminate if the number of axes on the page is an odd number.

The two time series are plotted on alternating axes with the data from one time series being plotted in black on one axis and the data from the other time series being plotted in red on the axis below that axis.

The earliest data is plotted on the pair of axes nearest the top of the Page moving from left to right across the page. Positive data values are plotted above the axis and negative values are plotted below the axis. When the right end of an axis is reached, the next data value is plotted on the left end of the second axis below it skipping one axis in the process. When the right end of the last pair of axes on the Page is reached, a new Page object is created and the next pair of data values are plotted at the left end of the top pair of axes on that new Page object.

As mentioned above, there are two overloaded versions of the constructor for the PlotALot03 class. One overloaded version accepts several incoming parameters allowing the user to control various aspects of the plotting format. A second overloaded version accepts a title string only

Listing 37. PlotALot03.java.

and sets all of the plotting parameters to default values. You can easily modify these default values and recompile the class if you prefer different default values.

The parameters for the version of the constructor that accepts plotting format information are:

String title: Title for the Frame object. This title is concatenated with the page number and the result appears in the banner at the top of the Frame.

int frameWidth: The Frame width in pixels.

int frameHeight: The Frame height in pixels.

int traceSpacing: Distance between trace axes in pixels.

int sampSpace: Number of pixels dedicated to each data sample in pixels per sample. Must be 1 or greater.

int ovalWidth: Width of an oval that is used to mark each sample value on the plot.

int ovalHeight: Height of an oval that is used to mark each sample value on the plot.

For test purposes, the main method instantiates a single plotting object and feeds two time series to that plotting object. Plotting parameters are specified for the plotting object by using the overloaded version of the constructor that accepts plotting parameters.

The data that is fed to the plotting object is white random noise. One of the time series is the sequence of values obtained from a random number generator. The other time series is the same as the first. Thus, the pairs of black and

Listing 37. PlotALot03.java.

red time series that are plotted should have the same shape making it easy to confirm that the process of plotting the two time series is behaving the same in both cases.

Fifteen of the data values for each time series are not random. Seven of the values for each of the time series are set to values of 0,0,25,-25,25,0,0. This is done to confirm the proper transition from the end of one page to the beginning of the next page.

In addition, eight of the values for each time series are set to 0,0,20,20,-20,-20,0,0. This is done in order to confirm the proper transition from one trace to the next trace on the same page.

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly. Note, however that these are the correct locations for an AWT Frame object under WinXP. A Frame may have different inset values under other operating systems, which may cause these specific locations to be incorrect for that operating system. In that case, the values will be plotted but they won't confirm the proper transition.

The following information about the plotting parameters is displayed on the command line screen when the class is used for plotting. The values shown below result from the execution of the main method of the class for test purposes.

Listing 37. PlotALot03.java.

```
Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 90
```

There are two overloaded versions of the `plotData` method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. This version requires two parameters, which are coordinate values in pixels. The first parameter specifies the horizontal coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. The second parameter specifies the vertical coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. Specifying coordinate values of 0,0 causes the stack to be located in the upper left corner of the screen.

The other overloaded version of `plotData` places the stack of pages in the upper left corner of the screen by default. The main method in this class uses the second version causing the stack of pages to appear in the upper left corner of the screen by default.

Each page has a `WindowListener` that will terminate the program if the user clicks the close button on the Frame.

Listing 37. PlotALot03.java.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.

```
*****/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot03{
    //This main method is provided so that the
    // class can be run as an application to test
    // itself.
    public static void main(String[] args){
        //Instantiate a plotting object using the
        // version of the constructor that allows for
        // controlling the plotting parameters.
        PlotALot03 plotObjectA =
            new PlotALot03("A",158,270,36,5,4,4);

        //Feed pairs of data values to the plotting
        // object.
        for(int cnt = 0;cnt < 175;cnt++){
            //Plot some white random noise Note that
            // fifteen of the values for each time
            // series are not random. See the opening
            // comments for a discussion of the reasons
            // why. Cause the values for the second
            // time series to be the same as the
            // values for the first time series.
            double valBlack = (Math.random() - 0.5)*25;
            double valRed = valBlack;
            //Feed pairs of values to the plotting
            // object by calling the feedData method
            // once for each pair of data values.
            if(cnt == 87){
                plotObjectA.feedData(0,0);
            }
        }
    }
}
```

Listing 37. PlotALot03.java.

```
        }else if(cnt == 88){
            plotObjectA.feedData(0,0);
        }else if(cnt == 89){
            plotObjectA.feedData(25,25);
        }else if(cnt == 90){
            plotObjectA.feedData(-25,-25);
        }else if(cnt == 91){
            plotObjectA.feedData(25,25);
        }else if(cnt == 92){
            plotObjectA.feedData(0,0);
        }else if(cnt == 93){
            plotObjectA.feedData(0,0);
        }else if(cnt == 26){
            plotObjectA.feedData(0,0);
        }else if(cnt == 27){
            plotObjectA.feedData(0,0);
        }else if(cnt == 28){
            plotObjectA.feedData(20,20);
        }else if(cnt == 29){
            plotObjectA.feedData(20,20);
        }else if(cnt == 30){
            plotObjectA.feedData(-20,-20);
        }else if(cnt == 31){
            plotObjectA.feedData(-20,-20);
        }else if(cnt == 32){
            plotObjectA.feedData(0,0);
        }else if(cnt == 33){
            plotObjectA.feedData(0,0);
        }else{
            plotObjectA.feedData(valBlack,valRed);
        }
    } //end else
} //end for loop
//Cause the data to be plotted in the default
// screen location.
plotObjectA.plotData();
} //end main
```

Listing 37. PlotALot03.java.

```
//-----//

String title;
int frameWidth;
int frameHeight;
int traceSpacing;//pixels between traces
int sampSpacing;//pixels between samples
int ovalWidth;//width of sample marking oval
int ovalHeight;//height of sample marking oval

int tracesPerPage;
int samplesPerPage;
int pageCount = 0;
int sampleCounter = 0;
ArrayList <Page> pageLinks =
                                new ArrayList<Page>();

//There are two overloaded versions of the
// constructor for this class. This
// overloaded version accepts several incoming
// parameters allowing the user to control
// various aspects of the plotting format. A
// different overloaded version accepts a title
// string only and sets all of the plotting
// parameters to default values.
PlotALot03(String title,//Frame title
            int frameWidth,//in pixels
            int frameHeight,//in pixels
            int traceSpacing,//in pixels
            int sampSpace,//in pixels per sample
            int ovalWidth,//sample marker width
            int ovalHeight)//sample marker hite
{
    //constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1. Convert to
    // pixels between samples for purposes of
```

Listing 37. PlotALot03.java.

```
// computation.
this.title = title;
this.frameWidth = frameWidth;
this.frameHeight = frameHeight;
this.traceSpacing = traceSpacing;
//Convert to pixels between samples.
this.sampSpacing = sampSpace - 1;
this.ovalWidth = ovalWidth;
this.ovalHeight = ovalHeight;

//The following object is instantiated solely
// to provide information about the width and
// height of the canvas. This information is
// used to compute a variety of other
// important values.
Page tempPage = new Page(title);
int canvasWidth = tempPage.canvas.getWidth();
int canvasHeight =
    tempPage.canvas.getHeight();
//Display information about this plotting
// object.
System.out.println("\nTitle: " + title);
System.out.println(
    "Frame width: " + tempPage.getWidth());
System.out.println(
    "Frame height: " + tempPage.getHeight());
System.out.println(
    "Page width: " + canvasWidth);
System.out.println(
    "Page height: " + canvasHeight);
System.out.println(
    "Trace spacing: " + traceSpacing);
System.out.println(
    "Sample spacing: " + (sampSpacing + 1));
if(sampSpacing < 0){
    System.out.println("Terminating");
```

Listing 37. PlotALot03.java.

```
        System.exit(0);
    }//end if
    //Get rid of this temporary page.
    tempPage.dispose();
    //Now compute the remaining important values.
    tracesPerPage = canvasHeight/traceSpacing -
                    traceSpacing/2/traceSpacing;
    System.out.println("Traces per page: "
                      + tracesPerPage);
    if((tracesPerPage == 0) ||
        (tracesPerPage%2 != 0) ){
        System.out.println("Terminating program");
        System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                    (sampSpacing + 1)/2;
    System.out.println("Samples per page: "
                      + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
} //end constructor
//-----//

PlotALot03(String title){
    //call the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
} //end overloaded constructor
//-----//

//call this method once for each pair of data
// values to be plotted.
void feedData(double valBlack,double valRed){
```

Listing 37. PlotALot03.java.

```
    if((sampleCounter) == samplesPerPage){
        //if the page is full, increment the page
        // counter, create a new empty page, and
        // reset the sample counter.
        pageCounter++;
        sampleCounter = 0;
        pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the MyCanvas
    // object to be used later to paint the
    // screen. Then increment the sample
    // counter. The sample values pass through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
        valBlack, valRed, sampleCounter);
    sampleCounter++;
} //end feedData
//-----//

//There are two overloaded versions of the
// plotData method. One version allows the
// user to specify the location on the screen
// where the stack of plotted pages will
// appear. The other version places the stack
// in the upper left corner of the screen.

//call one of the overloaded versions of
// this method once when all data has been fed
// to the plotting object in order to rearrange
// the order of the pages with page 0 at the
// top of the stack on the screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen. Values of 0,0 will
```


Listing 37. PlotALot03.java.

```
// place the stack at the upper left corner of
// the screen. Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
    Page lastPage =
        pageLinks.get(pageLinks.size() - 1);
    //Delay until last page becomes visible.
    while(!lastPage.isVisible()){
        //Loop until last page becomes visible
    }//end while loop

    Page tempPage = null;
    //Make all pages invisible
    for(int cnt = 0;cnt < (pageLinks.size());
        cnt++){
        tempPage = pageLinks.get(cnt);
        tempPage.setVisible(false);
    }//end for loop

    //Now make all pages visible in reverse order
    // so that page 0 will be on top of the
    // stack on the screen.
    for(int cnt = pageLinks.size() - 1;cnt >= 0;
        cnt--){
        tempPage = pageLinks.get(cnt);
        tempPage.setLocation(xCoor,yCoor);
        tempPage.setVisible(true);
    }//end for loop

}//end plotData(int xCoor,int yCoor)
//-----//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
```

Listing 37. PlotALot03.java.

```
void plotData(){
    plotData(0,0); //call overloaded version
} //end plotData()
//-----//

//Inner class. A PlotALot03 object may
// have as many Page objects as are required
// to plot all of the data values. The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot03 object.
class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){ //constructor
        canvas = new MyCanvas();
        add(canvas);
        setSize(frameWidth,frameHeight);
        setTitle(title + " Page: " + pageCounter);
        setVisible(true);

        //-----//
        //Anonymous inner class to terminate the
        // program when the user clicks the close
        // button on the Frame.
        addWindowListener(
            new WindowAdapter(){
                public void windowClosing(
                    WindowEvent e){
                    System.exit(0); //terminate program
                } //end windowClosing()
            } //end WindowAdapter
        ); //end addWindowListener
        //-----//
    } //end constructor
```

Listing 37. PlotALot03.java.

```
//=====//

//This method receives a pair of sample
// values of type double and stores each of
// them in a separate array object belonging
// to the MyCanvas object.
void putData(double valBlack,double valRed,
              int sampleCounter){
    canvas.blackData[sampleCounter] = valBlack;
    canvas.redData[sampleCounter] = valRed;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
} //end putData

//=====//
//Inner class
class MyCanvas extends Canvas{
    double [] blackData =
        new double[samplesPerPage];
    double [] redData =
        new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
        //Draw horizontal axes, one for each
        // trace.
        for(int cnt = 0;cnt < tracesPerPage;
            cnt++){
            g.drawLine(0,
                (cnt+1)*traceSpacing,
                this.getWidth(),
```

Listing 37. PlotALot03.java.

```
                (cnt+1)*traceSpacing);
    } //end for loop

    //Plot the points if there are any to be
    // plotted.
    if(sampleCounter > 0){
        for(int cnt = 0; cnt <= sampleCounter;
            cnt++){

            //Begin by plotting the values from
            // the blackData array object.
            g.setColor(Color.BLACK);

            //Compute a vertical offset to locate
            // the black data on the odd numbered
            // axes on the page.
            int yOffset =
                ((1 + cnt*(sampSpacing + 1)/
                    this.getWidth())*2*traceSpacing)
                    - traceSpacing;

            //Draw an oval centered on the sample
            // value to mark the sample in the
            // plot. It is best if the dimensions
            // of the oval are evenly divisible
            // by 2 for centering purposes.
            //Reverse the sign of the sample
            // value to cause positive sample
            // values to be plotted above the
            // axis.

            g.drawOval(cnt*(sampSpacing + 1)%
                this.getWidth() - ovalWidth/2,
                yOffset - (int)blackData[cnt]
                    - ovalHeight/2,
                ovalWidth,
```

Listing 37. PlotALot03.java.

```
        ovalHeight);

    //Connect the sample values with
    // straight lines. Do not draw a
    // line connecting the last sample in
    // one trace to the first sample in
    // the next trace.
    if(cnt*(sampSpacing + 1)%
        this.getWidth() >=
        sampSpacing + 1){
        g.drawLine(
            (cnt - 1)*(sampSpacing + 1)%
                this.getWidth(),
            yOffset - (int)blackData[cnt-1],
            cnt*(sampSpacing + 1)%
                this.getWidth(),
            yOffset - (int)blackData[cnt]);
    }//end if

    //Now plot the data stored in the
    // redData array object.
    g.setColor(Color.RED);
    //Compute a vertical offset to locate
    // the red data on the even numbered
    // axes on the page.
    yOffset = (1 + cnt*(sampSpacing + 1)/
        this.getWidth())*2*traceSpacing;

    //Draw the ovals as described above.
    g.drawOval(cnt*(sampSpacing + 1)%
        this.getWidth() - ovalWidth/2,
        yOffset - (int)redData[cnt]
            - ovalHeight/2,
        ovalWidth,
        ovalHeight);
```

Listing 37. PlotALot03.java.

```
        //Connect the sample values with
        // straight lines as described above.
        if(cnt*(sampSpacing + 1)%
            this.getWidth() >=
            sampSpacing + 1){
            g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                    this.getWidth(),
                yOffset - (int)redData[cnt-1],
                cnt*(sampSpacing + 1)%
                    this.getWidth(),
                yOffset - (int)redData[cnt]);

            }//end if
        }//end for loop
    }//end if for sampleCounter > 0
} //end overridden paint method
} //end inner class MyCanvas
} //end inner class Page
} //end class PlotALot03
//=====//
```

Listing 38. PlotALot04.java.

```
/*File PlotALot04.java
Copyright 2005, R.G.Baldwin
This program is an update to the program named
PlotALot03. This program is designed to plot
large amounts of time-series data for three
```

Listing 38. PlotALot04.java.

channels on separate horizontal axes. One set of data is plotted using the color black. The second set of data is plotted using the color red. The third set of data is plotted using the color blue.

See PlotALot03 for a class that plots two channels of data in black and red on alternating axes.

See PlotALot02 for a class that plots two channels of data in black and red superimposed on the same axes.

See PlotALot01.java for a one-channel program.

The class provides a main method so that the class can be run as an application to test itself.

There are three steps involved in the use of this class for plotting time series data:

1. Instantiate a plotting object of type PlotALot04 using one of two overloaded constructors.
2. Feed triplets of data values that are to be plotted to the plotting object by calling the feedData method once for each triplet of data values. The first value in the triplet will be plotted in black on one axis. The second value in the triplet will be plotted in red on an axis below that axis. The third value in the triplet will be plotted in blue on an axis below that one.
3. call one of two overloaded plotData methods on the plotting object once all of the data

Listing 38. PlotALot04.java.

has been fed to the object. This causes all of the data to be plotted.

A using program can instantiate as many plotting objects as are needed to plot all of the different triplets of data that need to be plotted. Each plotting object can be used to plot as many triplets of data values as need be plotted until the program runs out of available memory.

The plotting object of type PlotALot04 owns one or more Page objects that extend the Frame class. The plotting object can own as many Page objects as are necessary to plot all of the triplets of data that are fed to that plotting object.

The program produces a graphic output consisting of a stack of Page objects on the screen, with the data plotted on a Canvas object contained by the Page object. The Page showing the earliest data is on the top of the stack and the Page showing the latest data is on the bottom of the stack. The Page objects on the top of the stack must be physically moved in order to see the Page objects on the bottom of the stack.

Each Page object contains three or more horizontal axes on which the data is plotted. The program will terminate if the number of axes on the page is not evenly divisible by 3.

The three time series are plotted on separate axes with the data from one time series being plotted in black on one axis, the data from the second time series being plotted in red on

Listing 38. PlotALot04.java.

the axis below that axis, and the data from the third time series being plotted in blue on the axis below that axis.

The earliest data is plotted on the three axes nearest the top of the Page moving from left to right across the page. Positive data values are plotted above the axis and negative values are plotted below the axis. When the right end of an axis is reached, the next data value is plotted on the left end of the third axis below it skipping two axes in the process. When the right end of the last triplet of axes on the Page is reached, a new Page object is created and the next triplet of data values are plotted at the left end of the top three axes on that new Page object.

As mentioned above, there are two overloaded versions of the constructor for the PlotALot04 class. One overloaded version accepts several incoming parameters allowing the user to control various aspects of the plotting format. A second overloaded version accepts a title string only and sets all of the plotting parameters to default values. You can easily modify these default values and recompile the class if you prefer different default values.

The parameters for the version of the constructor that accepts plotting format information are:

String title: Title for the Frame object. This title is concatenated with the page number and the result appears in the banner at the top of the Frame.

Listing 38. PlotALot04.java.

```
int frameWidth:The Frame width in pixels.  
int frameHeight: The Frame height in pixels.  
int traceSpacing: Distance between trace axes in  
pixels.  
int sampSpace: Number of pixels dedicated to each  
data sample in pixels per sample. Must be 1 or  
greater.  
int ovalWidth: Width of an oval that is used to  
mark each sample value on the plot.  
int ovalHeight: Height of an oval that is used to  
mark each sample value on the plot.
```

For test purposes, the main method instantiates a single plotting object and feeds three time series to that plotting object. Plotting parameters are specified for the plotting object by using the overloaded version of the constructor that accepts plotting parameters.

The data that is fed to the plotting object is white random noise. One of the time series is the sequence of values obtained from a random number generator. The other two time series are the same as the first. Thus, the triplets of black, red, and blue time series that are plotted should have the same shape making it easy to confirm that the process of plotting the three time series is behaving the same in all three cases.

Fifteen of the data values for each time series are not random. Seven of the values for each of the time series are set to values of 0,0,25,-25,25,0,0. This is done to confirm the proper transition from the end of one page to the beginning of the next page.

Listing 38. PlotALot04.java.

In addition, eight of the values for each time series are set to 0,0,20,20,-20,-20,0,0. This is done in order to confirm the proper transition from one trace to the next trace on the same page.

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly. Note, however that these are the correct locations for an AWT Frame object under WinXP. A Frame may have different inset values under other operating systems, which may cause these specific locations to be incorrect for that operating system. In that case, the values will be plotted but they won't confirm the proper transition.

The following information about the plotting parameters is displayed on the command line screen when the class is used for plotting. The values shown below result from the execution of the main method of the class for test purposes.

Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 60

There are two overloaded versions of the plotData

Listing 38. PlotALot04.java.

method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. This version requires two parameters, which are coordinate values in pixels. The first parameter specifies the horizontal coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. The second parameter specifies the vertical coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. Specifying coordinate values of 0,0 causes the stack to be located in the upper left corner of the screen.

The other overloaded version of `plotData` places the stack of pages in the upper left corner of the screen by default. The main method in this class uses the second version causing the stack of pages to appear in the upper left corner of the screen by default.

Each page has a `WindowListener` that will terminate the program if the user clicks the close button on the `Frame`.

The program was tested using J2SE 5.0 and WinXP. Requires J2SE 5.0 to support generics.

```
*****/  
  
import java.awt.*;  
import java.awt.event.*;  
import java.util.*;  
  
public class PlotALot04{  
    //This main method is provided so that the  
    // class can be run as an application to test
```

Listing 38. PlotALot04.java.

```
// itself.
public static void main(String[] args){
    //Instantiate a plotting object using the
    // version of the constructor that allows for
    // controlling the plotting parameters.
    PlotALot04 plotObjectA =
        new PlotALot04("A",158,270,36,5,4,4);

    //Feed triplets of data values to the
    // plotting object.
    for(int cnt = 0;cnt < 115;cnt++){
        //Plot some white random noise. Note that
        // fifteen of the values for each time
        // series are not random. See the opening
        // comments for a discussion of the reasons
        // why.
        double valBlack = (Math.random() - 0.5)*25;
        double valRed = valBlack;
        double valBlue = valBlack;
        //Feed triplets of values to the plotting
        // object by calling the feedData method
        // once for each triplet of data values.
        if(cnt == 57){
            plotObjectA.feedData(0,0,0);
        }else if(cnt == 58){
            plotObjectA.feedData(0,0,0);
        }else if(cnt == 59){
            plotObjectA.feedData(25,25,25);
        }else if(cnt == 60){
            plotObjectA.feedData(-25,-25,-25);
        }else if(cnt == 61){
            plotObjectA.feedData(25,25,25);
        }else if(cnt == 62){
            plotObjectA.feedData(0,0,0);
        }else if(cnt == 63){
            plotObjectA.feedData(0,0,0);
        }
    }
}
```

Listing 38. PlotALot04.java.

```
        }else if(cnt == 26){
            plotObjectA.feedData(0,0,0);
        }else if(cnt == 27){
            plotObjectA.feedData(0,0,0);
        }else if(cnt == 28){
            plotObjectA.feedData(20,20,20);
        }else if(cnt == 29){
            plotObjectA.feedData(20,20,20);
        }else if(cnt == 30){
            plotObjectA.feedData(-20,-20,-20);
        }else if(cnt == 31){
            plotObjectA.feedData(-20,-20,-20);
        }else if(cnt == 32){
            plotObjectA.feedData(0,0,0);
        }else if(cnt == 33){
            plotObjectA.feedData(0,0,0);
        }else{
            plotObjectA.feedData(valBlack,
                                valRed,
                                valBlue);
        }
    }//end else
} //end for loop
//Cause the data to be plotted in the default
// screen location.
plotObjectA.plotData();
} //end main
//-----//

String title;
int framewidth;
int frameHeight;
int traceSpacing;//pixels between traces
int sampSpacing;//pixels between samples
int ovalWidth;//width of sample marking oval
int ovalHeight;//height of sample marking oval
```

Listing 38. PlotALot04.java.

```
int tracesPerPage;
int samplesPerPage;
int pageCounter = 0;
int sampleCounter = 0;
ArrayList <Page> pageLinks =
                                new ArrayList<Page>();

//There are two overloaded versions of the
// constructor for this class. This
// overloaded version accepts several incoming
// parameters allowing the user to control
// various aspects of the plotting format. A
// different overloaded version accepts a title
// string only and sets all of the plotting
// parameters to default values.
PlotALot04(String title,//Frame title
            int frameWidth,//in pixels
            int frameHeight,//in pixels
            int traceSpacing,//in pixels
            int sampSpace,//in pixels per sample
            int ovalWidth,//sample marker width
            int ovalHeight)//sample marker hite
{
    //constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1. Convert to
    // pixels between samples for purposes of
    // computation.
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between samples.
    this.sampSpacing = sampSpace - 1;
    this.ovalWidth = ovalWidth;
    this.ovalHeight = ovalHeight;
}
```

Listing 38. PlotALot04.java.

```
//The following object is instantiated solely
// to provide information about the width and
// height of the canvas. This information is
// used to compute a variety of other
// important values.
Page tempPage = new Page(title);
int canvasWidth = tempPage.canvas.getWidth();
int canvasHeight =
    tempPage.canvas.getHeight();
//Display information about this plotting
// object.
System.out.println("\nTitle: " + title);
System.out.println(
    "Frame width: " + tempPage.getWidth());
System.out.println(
    "Frame height: " + tempPage.getHeight());
System.out.println(
    "Page width: " + canvasWidth);
System.out.println(
    "Page height: " + canvasHeight);
System.out.println(
    "Trace spacing: " + traceSpacing);
System.out.println(
    "Sample spacing: " + (sampSpacing + 1));
if(sampSpacing < 0){
    System.out.println("Terminating");
    System.exit(0);
} //end if
//Get rid of this temporary page.
tempPage.dispose();
//Now compute the remaining important values.
tracesPerPage = canvasHeight/traceSpacing -
    traceSpacing/2/traceSpacing;
System.out.println("Traces per page: "
    + tracesPerPage);
if((tracesPerPage == 0) ||
```


Listing 38. PlotALot04.java.

```

                                (tracesPerPage%3 != 0) ){
        System.out.println("Terminating program");
        System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                    (sampSpacing + 1)/3;
    System.out.println("Samples per page: "
                      + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
} //end constructor
//-----//

PlotALot04(String title){
    //call the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
} //end overloaded constructor
//-----//

//call this method once for each triplet of
// data values to be plotted.
void feedData(double valBlack,
              double valRed,
              double valBlue){
    if((sampleCounter) == samplesPerPage){
        //if the page is full, increment the page
        // counter, create a new empty page, and
        // reset the sample counter.
        pageCounter++;
        sampleCounter = 0;
        pageLinks.add(new Page(title));
    } //end if
}
```

Listing 38. PlotALot04.java.

```
//Store the sample values in the MyCanvas
// object to be used later to paint the
// screen. Then increment the sample
// counter. The sample values pass through
// the page object into the current MyCanvas
// object.
pageLinks.get(pageCounter).putData(
                                valBlack,
                                valRed,
                                valBlue,
                                sampleCounter);

    sampleCounter++;
} //end feedData
//-----//

//There are two overloaded versions of the
// plotData method. One version allows the
// user to specify the location on the screen
// where the stack of plotted pages will
// appear. The other version places the stack
// in the upper left corner of the screen.

//call one of the overloaded versions of
// this method once when all data has been fed
// to the plotting object in order to rearrange
// the order of the pages with page 0 at the
// top of the stack on the screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen. Values of 0,0 will
// place the stack at the upper left corner of
// the screen. Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
```

Listing 38. PlotALot04.java.

```
    Page lastPage =
        pageLinks.get(pageLinks.size() - 1);
    //Delay until last page becomes visible.
    while(!lastPage.isVisible()){
        //Loop until last page becomes visible
    }//end while loop

    Page tempPage = null;
    //Make all pages invisible
    for(int cnt = 0;cnt < (pageLinks.size());
        cnt++){
        tempPage = pageLinks.get(cnt);
        tempPage.setVisible(false);
    }//end for loop

    //Now make all pages visible in reverse order
    // so that page 0 will be on top of the
    // stack on the screen.
    for(int cnt = pageLinks.size() - 1;cnt >= 0;
        cnt--){
        tempPage = pageLinks.get(cnt);
        tempPage.setLocation(xCoor,yCoor);
        tempPage.setVisible(true);
    }//end for loop

}//end plotData(int xCoor,int yCoor)
//-----//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
void plotData(){
    plotData(0,0);//call overloaded version
}//end plotData()
//-----//
```

Listing 38. PlotALot04.java.

```
//Inner class. A PlotALot04 object may
// have as many Page objects as are required
// to plot all of the data values. The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot04 object.
class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
        canvas = new MyCanvas();
        add(canvas);
        setSize(frameWidth,frameHeight);
        setTitle(title + " Page: " + pageCounter);
        setVisible(true);

        //-----//
        //Anonymous inner class to terminate the
        // program when the user clicks the close
        // button on the Frame.
        addWindowListener(
            new WindowAdapter(){
                public void windowClosing(
                    WindowEvent e){
                    System.exit(0);//terminate program
                }//end windowClosing()
            }//end WindowAdapter
        );//end addWindowListener
        //-----//
    }//end constructor
    //=====//

    //This method receives a triplet of sample
    // values of type double and stores each of
    // them in a separate array object belonging
```

Listing 38. PlotALot04.java.

```
// to the MyCanvas object.
void putData(double valBlack,
             double valRed,
             double valBlue,
             int sampleCounter){
    canvas.blackData[sampleCounter] = valBlack;
    canvas.redData[sampleCounter] = valRed;
    canvas.blueData[sampleCounter] = valBlue;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
} //end putData

//=====//
//Inner class
class MyCanvas extends Canvas{
    double [] blackData =
        new double[samplesPerPage];
    double [] redData =
        new double[samplesPerPage];
    double [] blueData =
        new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
        //Draw horizontal axes, one for each
        // trace.
        for(int cnt = 0; cnt < tracesPerPage;
            cnt++){
            g.drawLine(0,
                (cnt+1)*traceSpacing,
                this.getWidth(),
```

Listing 38. PlotALot04.java.

```
                (cnt+1)*traceSpacing);
    } //end for loop

    //Plot the points if there are any to be
    // plotted.
    if(sampleCounter > 0){
        for(int cnt = 0; cnt <= sampleCounter;
            cnt++){

            //Begin by plotting the values from
            // the blackData array object.
            g.setColor(Color.BLACK);

            //Compute a vertical offset to locate
            // the black data on every third axis
            // on the page.
            int yOffset =
                ((1 + cnt*(sampSpacing + 1)/
                    this.getWidth())*3*traceSpacing)
                - 2*traceSpacing;

            //Draw an oval centered on the sample
            // value to mark the sample in the
            // plot. It is best if the dimensions
            // of the oval are evenly divisible
            // by 2 for centering purposes.
            //Reverse the sign of the sample
            // value to cause positive sample
            // values to be plotted above the
            // axis.

            g.drawOval(cnt*(sampSpacing + 1)%
                this.getWidth() - ovalWidth/2,
                yOffset - (int)blackData[cnt]
                    - ovalHeight/2,
                ovalWidth,
```

Listing 38. PlotALot04.java.

```
        ovalHeight);

    //Connect the sample values with
    // straight lines. Do not draw a
    // line connecting the last sample in
    // one trace to the first sample in
    // the next trace.
    if(cnt*(sampSpacing + 1)%
        this.getWidth() >=
        sampSpacing + 1){
        g.drawLine(
            (cnt - 1)*(sampSpacing + 1)%
                this.getWidth(),
            yOffset - (int)blackData[cnt-1],
            cnt*(sampSpacing + 1)%
                this.getWidth(),
            yOffset - (int)blackData[cnt]);
    }//end if

    //Now plot the data stored in the
    // redData array object.
    g.setColor(Color.RED);
    //Compute a vertical offset to locate
    // the red data on every third axis
    // on the page.
    yOffset = (1 + cnt*(sampSpacing + 1)/
        this.getWidth())*3*traceSpacing
        - traceSpacing;

    //Draw the ovals as described above.
    g.drawOval(cnt*(sampSpacing + 1)%
        this.getWidth() - ovalWidth/2,
        yOffset - (int)redData[cnt]
            - ovalHeight/2,
        ovalWidth,
        ovalHeight);
```

Listing 38. PlotALot04.java.

```
//Connect the sample values with
// straight lines as described above.
if(cnt*(sampSpacing + 1)%
    this.getWidth() >=
    sampSpacing + 1){
    g.drawLine(
        (cnt - 1)*(sampSpacing + 1)%
            this.getWidth(),
        yOffset - (int)redData[cnt-1],
        cnt*(sampSpacing + 1)%
            this.getWidth(),
        yOffset - (int)redData[cnt]);
}

//end if

//Now plot the data stored in the
// blueData array object.
g.setColor(Color.BLUE);
//Compute a vertical offset to locate
// the blue data on every third axis
// on the page.
yOffset = (1 + cnt*(sampSpacing + 1)/
    this.getWidth())*3*traceSpacing;

//Draw the ovals as described above.
g.drawOval(cnt*(sampSpacing + 1)%
    this.getWidth() - ovalWidth/2,
    yOffset - (int)blueData[cnt]
        - ovalHeight/2,
    ovalWidth,
    ovalHeight);

//Connect the sample values with
// straight lines as described above.
```


Listing 38. PlotALot04.java.

```
        if(cnt*(sampSpacing + 1)%
                               this.getWidth() >=
                               sampSpacing + 1){
            g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                               this.getWidth(),
                yOffset - (int)blueData[cnt-1],
                cnt*(sampSpacing + 1)%
                               this.getWidth(),
                yOffset - (int)blueData[cnt]);
        }//end if
    }//end for loop
} //end if for sampleCounter > 0
} //end overridden paint method
} //end inner class MyCanvas
} //end inner class Page
} //end class PlotALot04
//=====//
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1492-Plotting Large Quantities of Data using Java
- File: Java1492.htm
- Published: 08/23/15

Learn how to use Java to plot millions of multi-channel data values in an easy-to-view format with very little programming effort.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1478-Fun with Java, How and Why Spectral Analysis Works
Baldwin explains how the Fourier transform can be used to determine the spectral content of a signal in the time domain.

Revised: Fri Oct 16 23:16:39 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Multiply-add operations](#)
 - [Computational requirements for DSP](#)
 - [Typical notation](#)
 - [A sum-of-products operation](#)
 - [An alternative notation](#)
- [Preview](#)
 - [What is a time series?](#)
 - [A new time series is produced](#)
 - [Calculation of the mean or average](#)

- [The Fourier transform](#)
 - [Time domain and frequency domain](#)
 - [Example of time domain and frequency domain](#)
 - [Forward and inverse transforms](#)
 - [Sampled time series](#)
 - [Integration and summation](#)
 - [The FFT algorithms](#)
 - [DFT versus FFT](#)
 - [The DFT algorithm](#)
 - [Why does this work?](#)
 - [Products of sine and cosine functions](#)
 - [Product of two sine functions having the same frequency](#)
 - [Product of two cosine functions having the same frequency](#)
 - [Product of a sine function and a cosine function](#)
 - [Neither sine nor cosine](#)
 - [What about non-matching frequency components?](#)
 - [Sum and difference frequencies](#)
 - [A form of measurement error](#)
 - [Product of two sine functions at different frequencies](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)

Preface

Programming in Java doesn't have to be dull and boring. In fact, it's possible to have a lot of fun while programming in Java. This module is one in a series that concentrates on having fun while programming in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the

Figures while you are reading about them.

Figures

- [Figure 1.](#) A typical sum-of-products operation.
- [Figure 2.](#) Alternative notation for a sum-of-products operation.
- [Figure 3.](#) Plot of values in a time series.
- [Figure 4.](#) Area under a periodic curve.
- [Figure 5.](#) Area under a periodic curve with an offset.
- [Figure 6.](#) Forward Fourier transform.
- [Figure 7.](#) Three trigonometric identities.
- [Figure 8.](#) Products of sine and cosine functions.
- [Figure 9.](#) Rewrite and simplify.
- [Figure 10.](#) Plot of $\sin(x)$ and $\sin(x)*\sin(x)$.
- [Figure 11.](#) Plot of $\cos(x)$ and $\cos(x)*\cos(x)$.
- [Figure 12.](#) Plot of $\sin(x)$, $\cos(x)$, and $\sin(x)*\cos(x)$.
- [Figure 13.](#) Products of sine and cosine functions.
- [Figure 14.](#) Plot of $\sin(1.8x)*\sin(2.2x)$.
- [Figure 15.](#) Plot of $\cos(1.8x)*\cos(2.2x)$.
- [Figure 16.](#) Plot of $\sin(1.8x)*\cos(2.2x)$.

Multiply-add operations

This module deals with a topic commonly known as Digital Signal Processing, (*DSP for short*) .

Computational requirements for DSP

The computational requirements for implementing DSP in a computer program are usually straightforward. Almost all DSP operations consist of multiplying corresponding values contained in two numeric series and then calculating the sum of the products. Sometimes, the final sum is divided by the total number of values included in the sum to produce an average. This is often referred to as a *sum-of-products* or *multiply-add operation* .

(This is the digital equivalent of integrating the product of two continuous functions between specified limits.)

Typical notation

Such an operation can be indicated by the symbolic notation shown in [Figure 1](#) (where the strange looking thing constructed of straight lines is the Greek letter sigma) .

Figure 1. A typical sum-of-products operation.

$$z = (1/N) * \sum_{n=0}^{N-1} x(n) * y(n)$$

A sum-of-products operation

This notation means that a new value for z is calculated by multiplying the first N corresponding samples from each of two numeric series (x(n) and y(n)), calculating the sum of the products, and dividing the sum by N.

(In this module, I will be dealing primarily with numeric series that represent samples from a continuous function taken over time. Therefore, I will often refer to the numeric series as a time series.)

An alternative notation

The above notation requires about six lines of text to construct, and therefore could easily become scrambled during the HTML publishing process. I have invented an alternative notation that means exactly the same thing, but is less likely to be damaged during the publishing process. My new notation is shown in [Figure 2](#). You should be able to compare this notation with [Figure 1](#) and correlate the terms in the notation to the verbal description of the operation given above.

Figure 2. Alternative notation for a sum-of-products operation.

$$z = (1/N) * S(n=0, N-1) [(x(n) * y(n))]$$

This is the notation that I will use in this module.

Preview

What is a time series?

I discussed the concept of a *time series* in some detail in my module titled [Dsp00104-Sampled Time Series](#). For purposes of this module, suffice it to

say that a time series is a set of sample values taken from a continuous function at equal increments of time over a specified time interval. For example, if you were to record the temperature in your office every minute for six hours, the set of 360 different values that you would record could be considered as a time series.

A new time series is produced

In DSP, we often multiply two time series together on a sample by sample basis. When I multiply the values in the time series $x(n)$ by the corresponding values in the time series $y(n)$, that produces a new time series, which I will call $w(n)$.

Calculation of the mean or average

If I compute the sum of the individual values in the series $w(n)$, and then divide that sum by the number of samples, this is nothing more than the calculation of the mean or average value of the time series named $w(n)$. Most DSP operations boil down to nothing more complicated than calculating the average value of the product of two time series.

Knowing what to multiply, when, and why

The real trick in DSP is knowing what to multiply, when to multiply it, and why to multiply it.

Some DSP algorithms are very complex. For example, the *Fast Fourier Transform (FFT)* algorithm, involves nothing more than a lot of multiply-add operations under the control of an extremely complex and efficient control structure.

In this module, I will concentrate on the *Discrete Fourier Transform (DFT)* algorithm, which is much less complex and therefore much easier to understand.

(While the DFT and the FFT produce the same results, the DFT typically runs much more slowly than the FFT, which is optimized for speed.)

Multiply-add speed is important

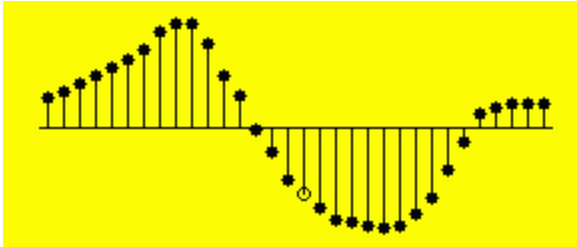
Some DSP processes require extremely large numbers of multiply-add operations. In order to perform DSP in real time, the equipment used to perform the arithmetic must be extremely fast. That is where the special DSP chips, *(which are designed to perform multiply-add operations at an extremely high rate of speed)* earn their keep.

The net area under the curve

If you plot a time series as a curve on a graph, as shown in [Figure 3](#), the sum of the values that make up the time series is an estimate of the net area under the curve.

(Assuming that the horizontal axis represents a value of zero, the sample values above the axis contribute a positive value to the net area and the sample values below the curve contribute a negative value to the net area. In the case of [Figure 3](#), I attempted to come up with a set of sample values that would produce a net area of zero. In other words, the area above the horizontal axis was intended to perfectly balance the area below the horizontal axis.)

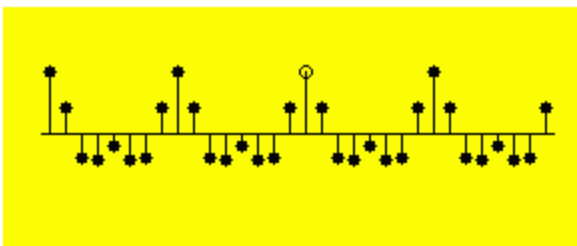
Figure 3. Plot of values in a time series.



A periodic example

A periodic time series is one in which a set of sample values repeats over time, provided that you record enough samples to include one or more periods. [Figure 4](#) shows a plot of a periodic time series. You can see that the same set of values repeats as you move from left to right on the curve plotted in [Figure 4](#).

Figure 4. Area under a periodic curve.



The sum of two curves

Periodic curves can often be viewed as the sum of two curves. One of the curves is the periodic component having a zero net area under the curve when measured across an even number of cycles. The other component is a constant bias offset that is added to every value of the periodic curve.

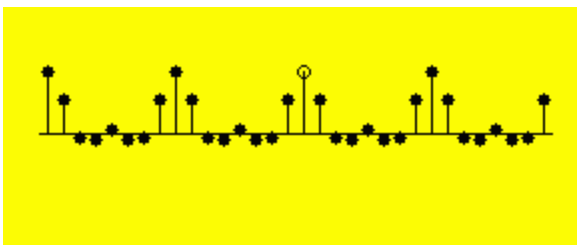
Each of the solid dark blobs in [Figure 4](#) is a sample value. The horizontal line represents a sample value of zero. (*The empty circle is the sample value half way through the sampling interval. The only reason it is different is to mark the mid point.*)

The net area under the curve

What is the net area under the curve in [Figure 4](#)? Can you examine the curve and come up with a good estimate. As it turns out, the net area under the curve in [Figure 4](#) is very close to zero (*at least it is as close to zero as I was able to draw it*).

Now take a look at [Figure 5](#). What is the net area under the curve in [Figure 5](#)?

Figure 5. Area under a periodic curve with an offset.



Compare [Figure 5](#) to [Figure 4](#)

Each of these curves describes the same periodic shape (*although [Figure 4](#) has a larger peak-to-peak amplitude, meaning simply that every value in [Figure 4](#) has been multiplied by the same scale factor*).

However, the curve in [Figure 5](#) is riding up on a positive bias, while the curve in [Figure 4](#) is centered about the horizontal axis. While the net area under the curve in [Figure 4](#) is near zero, the net area under the curve in [Figure 5](#) is a non-zero positive value.

The curve in [Figure 5](#) can be considered to consist of the sum of two parts. One part is a straight horizontal line on the positive side of the horizontal axis. The other part is the periodic curve from [Figure 4](#), added to that positive bias.

(The curve in [Figure 4](#) can also be considered to consist of the sum of two parts. However, in [Figure 4](#), the bias value is zero.)

The net area

The net area contributed by the periodic part of the curve in [Figure 5](#) continues to be zero. In effect, the non-zero net area under the curve in [Figure 5](#) is the amount of the positive bias multiplied by the number of points. In other words, because I captured an even number of cycles of the periodic portion of the curve in my calculation of net area, only the bias contributes a non-zero value to the net area under the total curve.

Part of a cycle

Had I failed to capture an even number of cycles of the periodic portion of the curve, then the positive lobes might not completely cancel the negative lobes, and the net area under the curve would be influenced by the periodic portion of the curve in addition to the bias.

Why is this important?

These concepts are extremely important in this module as we learn how to do frequency spectral analysis.

As you will see in this module, in doing frequency spectral analysis, we will form a product between a target time series and a sinusoid. The purpose is to measure the power contained in the time series at the frequency of the sinusoid.

The product of the target time series and the sinusoid will produce the sum of a potentially infinite number of periodic functions, some of which may be riding on a positive or negative bias. We will measure the amount of bias by computing the average value of the product time series. The amount of bias will be our estimate of the power contained in the target time series at the frequency of the sinusoid.

The Fourier transform

There is a mathematical process, known as the Fourier transform, which can be used to linearly transform information back and forth between two different domains. The information can be represented by sets of complex numbers in either or both domains.

The domains can represent a variety of different things. In DSP, the domains are often referred to as the *time domain* and the *frequency domain*, but that is not a requirement. For example, one of the domains could represent the samples that make up the pixels in a photograph and the other domain could represent something having to do with the manipulation of photographs.

Usually when dealing with the time domain and the frequency domain, the values that make up the samples in the time domain are purely real while the values that make up the samples in the frequency domain are complex containing both real and imaginary parts.

Time domain and frequency domain

For some purposes, it is preferable to have your information in the time domain. For other purposes, it is preferable to have your information in the frequency domain. The Fourier transform allows you to transform your information back and forth between these two domains at will.

For example, it is possible to transform information from the time domain into the frequency domain, modify that information in the frequency domain, and then transform the modified information back into the time domain. This is one way to accomplish frequency filtering.

Example of time domain and frequency domain

If you were to draw a graph of the voltage impinging on the speaker coils on your stereo system over time, that would be a time series, which is a member of the time domain.

If you were to observe the lights dancing up and down on the front of your equalizer while the music is playing, you would be observing the same information presented in the frequency domain. Typically the lights on the left represent low frequencies or bass while the lights on the right side represent high frequencies or treble. Often there is a slider associated with each vertical group of lights that allows you to apply filters to emphasize certain parts of the frequency spectrum and to de-emphasize other parts of the frequency spectrum.

Forward and inverse transforms

There are two very similar forms of the Fourier transform. The forward transform is typically used to transform information from the time domain into the frequency domain. The inverse transform is typically used to transform information from the frequency domain back into the time domain.

Sampled time series

The theoretical Fourier transform is defined using integral calculus as applied to continuous functions. As a practical matter, in the digital world, we almost never deal with continuous functions. Rather, we deal with functions that have been reduced to a series of discrete numbers (or samples), which are the result of some discrete measurement system.

(As mentioned earlier, recording the temperature in your office once each minute for twenty-four hours would produce such a discrete series of numbers.)

Integration and summation

In many cases, the integration operation encountered in integral calculus can be *approximated* in the digital world by a summation operation using discrete data. That is the case with the Fourier transform. Thus, the *(simple)* summation form of the Fourier transform that is applied to a discrete time series is known as the **Discrete Fourier Transform**, or **DFT**.

The FFT algorithms

The DFT is a computationally intense operation. Given certain restrictions involving the number of values in the time series and the number of frequencies at which the spectral analysis will be performed, there is a special algorithm that can result in computational economy in performing the transform. The algorithms that are used to realize that economy are commonly referred to as **Fast Fourier Transform** or **FFT** algorithms.

DFT versus FFT

The DFT is more general than the FFT, but the FFT is much faster than the DFT. It is important to understand that these are simply two different algorithms for doing the same thing. Either can be used to produce the same results (*but as mentioned earlier, the FFT is somewhat more restricted as to the number of time-domain and frequency-domain samples*) .

Because the DFT algorithm is somewhat easier to understand than the FFT algorithm, and also more general, I will concentrate on the DFT algorithm to explain how and why the Fourier transform accomplishes what it accomplishes.

The DFT algorithm

Using my alternative notation described earlier in [Figure 2](#), the expressions that you must evaluate to determine the frequency spectral content of a target time series at a frequency F are shown in [Figure 6](#) (*note that I didn't bother to divide by N which is fairly common practice*) .

Figure 6. Forward Fourier transform.

$$\text{Real}(F) = \sum_{n=0, N-1} [x(n) * \cos(2\pi * F * n)]$$

$$\text{Imag}(F) = \sum_{n=0, N-1} [x(n) * \sin(2\pi * F * n)]$$

$$\text{ComplexAmplitude}(F) = \text{Real}(F) - j * \text{Imag}(F)$$

$$\text{Power}(F) = \text{Real}(F) * \text{Real}(F) + \text{Imag}(F) * \text{Imag}(F)$$

What does this really mean?

Before you panic, let me explain what this means in layman's terms. Given a time series, $x(n)$, you can determine if that time series contains a cosine component or a sine component at a given frequency, F , by doing the following:

- Create one new time series, $\cos(n)$, which is a cosine function with the frequency F .
- Create another new time series, $\sin(n)$, which is a sine function with the frequency F . (*The methods needed to create the cosine and sine time series are available in the **Math** class in the standard Java library.*)
- Multiply $x(n)$ by $\cos(n)$ and compute the sum of the products. Save this value, calling it $\text{Real}(F)$. This is an estimate of the amplitude, if any, of the cosine component with the matching frequency contained in the time series $x(n)$.
- Multiply $x(n)$ by $\sin(n)$ and compute the sum of the products. Save this value, calling it $\text{Imag}(F)$. This is an estimate of the amplitude, if any, of the sine component with the matching frequency contained in the time series $x(n)$.
- Consider the values for $\text{Real}(F)$ and $\text{Imag}(F)$ to be the real and imaginary parts of a complex number.
- Consider the sum of the squares of the real and imaginary parts to represent the power at that frequency in the time series.

It's that simple

That's all there is to it. For each frequency of interest, you can use this process to compute a complex number, $\text{Real}(F) - j\text{Imag}(F)$, which represents the component of that frequency in the target time series.

(The mathematicians in the audience probably prefer to use the symbol i instead of the symbol j to represent the imaginary part. The use of j for this purpose comes from my electrical engineering background.)

Similarly, you can compute the sum of the squares of the real and imaginary parts and consider that to be a measure of the power at that frequency in the time series.

(This is typically the value that you would see being displayed by one of the dancing vertical bars on the front of the equalizer on your stereo system.)

Normally we are interested in more than one frequency, so we would repeat the above procedure once for each frequency of interest.

(This would produce the set of values that you would likely see being displayed by all of the dancing vertical bars on the front of the equalizer on your stereo system.)

Why does this work?

This works because of the three trigonometric identities shown in [Figure 7](#).

Figure 7. Three trigonometric identities.

Figure 7. Three trigonometric identities.

1. $\sin(a) \cdot \sin(b) = (1/2) \cdot (\cos(a-b) - \cos(a+b))$
2. $\cos(a) \cdot \cos(b) = (1/2) \cdot (\cos(a-b) + \cos(a+b))$
3. $\sin(a) \cdot \cos(b) = (1/2) \cdot (\sin(a+b) + \sin(a-b))$

Although these identities apply to the products of sine and cosine values for single angles **a** and **b**, it is a simple matter to extend them to represent the products of time series consisting of sine and cosine functions. Such an extension is shown in [Figure 8](#).

Products of sine and cosine functions

In each of the three cases shown in [Figure 8](#), the function $f(n)$ is a time series produced by multiplying two other time series, which are either sine functions or cosine functions.

Figure 8. Products of sine and cosine functions.

1. $f(n) = \sin(a \cdot n) \cdot \sin(b \cdot n) = (1/2) \cdot (\cos((a-b) \cdot n) - \cos((a+b) \cdot n))$
2. $f(n) = \cos(a \cdot n) \cdot \cos(b \cdot n) = (1/2) \cdot (\cos((a-b) \cdot n) + \cos((a+b) \cdot n))$
3. $f(n) = \sin(a \cdot n) \cdot \cos(b \cdot n) = (1/2) \cdot (\sin((a+b) \cdot n) + \sin((a-b) \cdot n))$

Rewrite and simplify

[Figure 9](#) rewrites and simplifies these three functions for the special case where $\mathbf{a=b}$, taking into account the fact that $\cos(0) = 1$ and $\sin(0) = 0$.

Figure 9. Rewrite and simplify.

$$1. f(n) = \sin(a*n)*\sin(a*n) = (1/2) - \cos(2*a*n)/2$$

$$2. f(n) = \cos(a*n)*\cos(a*n) = (1/2) + \cos(2*a*n)/2$$

$$3. f(n) = \sin(a*n)*\cos(a*n) = \sin(2*a*n)/2$$

What can we learn from these identities?

First you need to recall that the average of the values describing any true sinusoid is zero when the average is computed over an even number of cycles of the sinusoid.

(A true sinusoid does not have a bias to prevent it from being centered on the horizontal axis.)

If a time series consists of the sum of two true sinusoids, then the average of the values describing that time series will be zero if the average is computed over an even number of cycles of both sinusoids, and very close

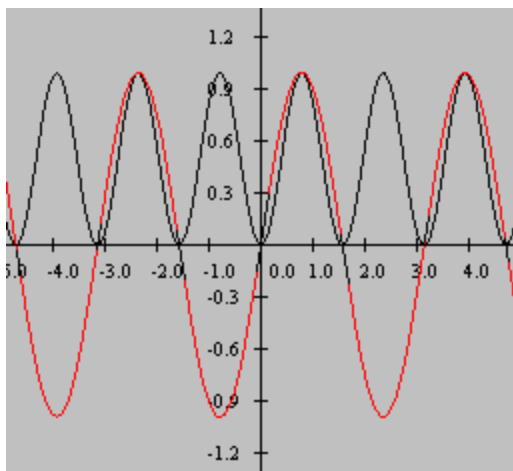
to zero if the average is computed over a period that is not an even number of cycles for either or both sinusoids.

(The average will approach zero as the length of data over which the average is computed increases.)

Product of two sine functions having the same frequency

Let's apply this knowledge to the three cases shown above for $\mathbf{a=b}$. Consider the time series for case 1 in [Figure 9](#) . This case is the product of two sine functions having the same frequency. The result of multiplying the two sine functions is shown graphically in [Figure 10](#) .

Figure 10. Plot of $\sin(x)$ and $\sin(x)*\sin(x)$.



The red curve in [Figure 10](#) shows the function $\sin(x)$, and the black curve shows the function produced by multiplying $\sin(x)$ by $\sin(x)$.

The sum of the product function is not zero

If you sum the values of the black curve over an even number of cycles, the sum will not be zero. Rather, it will be a positive, non-zero value.

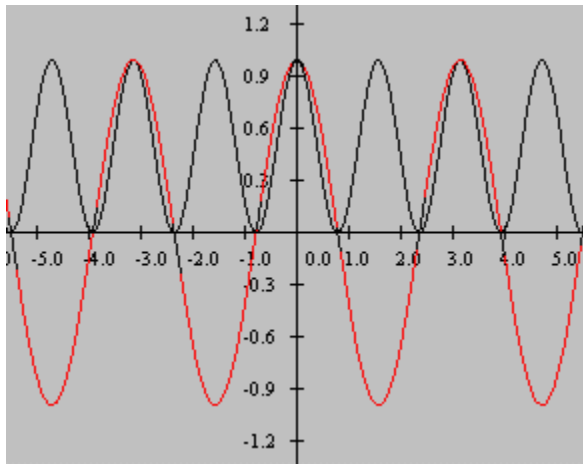
Now refer back to $\text{Imag}(F)$ in [Figure 6](#). The imaginary part is computed by multiplying the time series by a sine function and computing the sum of the products. If that time series contains a sine component with the same frequency as the sine function, that component will contribute a non-zero value to the sum of products. Thus, the imaginary part of the transform at that frequency will not be zero.

Product of two cosine functions having the same frequency

Now consider the time series for case 2 in [Figure 9](#). This case is the product of two cosine functions having the same frequency. The result of multiplying two cosine functions having the same frequency is shown graphically in [Figure 11](#).

Figure 11. Plot of $\cos(x)$ and $\cos(x)*\cos(x)$.

Figure 11. Plot of $\cos(x)$ and $\cos(x)*\cos(x)$.



The red curve in [Figure 11](#) shows the function $\cos(x)$, and the black curve shows the function produced by multiplying $\cos(x)$ by $\cos(x)$.

Again the sum of products is not zero

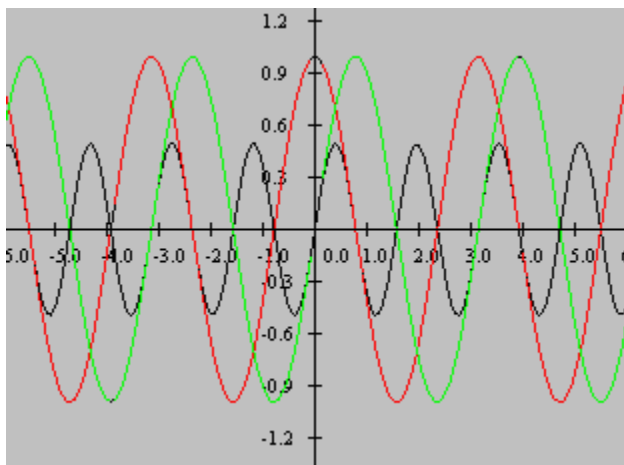
If you sum the values of the black curve in [Figure 11](#) over an even number of cycles, the sum will not be zero. Rather, it will be a positive, non-zero value.

Now refer back to the expression for $\text{Real}(F)$ in [Figure 6](#). The real part of the transform is computed by multiplying the time series by a cosine function having a particular frequency and computing the sum of products. If that time series contains a cosine component with the same frequency as the cosine function, that component will contribute a non-zero value to the sum of products. Thus, the real part of the transform at that frequency will not be zero.

Product of a sine function and a cosine function

Now consider the time series for case 3 in [Figure 9](#), which is the product of a sine function and a cosine function having the same frequency. The result of computing this product is shown graphically in [Figure 12](#)

Figure 12. Plot of $\sin(x)$, $\cos(x)$, and $\sin(x)*\cos(x)$.



The red curve in [Figure 12](#) shows the function $\cos(x)$, and the green curve shows the function $\sin(x)$. The black curve shows the function produced by multiplying $\sin(x)$ by $\cos(x)$.

The sum of the products will be zero

If you sum the values of the black curve over an even number of cycles, the sum will be zero.

Therefore, referring back to [Figure 6](#), we see that

- the $\text{Real}(F)$ computation measures only the cosine component in the time series at a particular frequency, and
- the $\text{Imag}(F)$ computation measures only the sine component in the time series having the same frequency.

The $\text{Real}(F)$ computation in [Figure 6](#) does not produce a non-zero output due to a sine component in the time series having the same frequency. The $\text{Imag}(F)$ computation in [Figure 6](#) does not produce a non-zero output due to a cosine component in the time series having the same frequency.

Thus, at a particular frequency, the existence of a cosine component in the target time series produces the real output, and the existence of a sine component in the target time series produces the imaginary output.

Neither sine nor cosine

In reality, the sinusoidal components that make up a time series will not usually be sine functions or cosine functions. Rather, they will be sinusoidal components having the same shape as a sine or cosine, but not having the same value at zero as either a sine function or a cosine function. However, it can be shown that a general sinusoidal function can always be represented by the sum of a sine function and a cosine function having different amplitudes and the same frequency.

(A proof of the above statement is beyond the scope of this module. You will simply have to accept on faith that a general time series can be represented as the sum of a potentially infinite number of sine functions and cosine functions of different frequencies and different amplitudes. It is these cosine and sine functions that constitute the real and imaginary components of the complex frequency spectrum.)

What about non-matching frequency components?

Now we know what happens when the frequency of the sin and cos terms in [Figure 6](#) match the frequency of sine and cosine components in the target time series. Another important question is, what do sine and cosine components in the target time series with frequencies different from the cos and sin terms in [Figure 6](#) contribute to the output?

The answer is not very much. Referring once more to the functional forms produced by multiplying sine and cosine functions (*repeated in [Figure 13](#) for convenience*), we see that for any values of **a** and **b**, where **a** is not equal to **b**, the function produced by multiplying two sinusoids will be the sum of two other sinusoids. The sum of the values for any sinusoid computed over an even number of cycles of the sinusoid will always be zero, and these sinusoids are no exception to that rule.

Figure 13. Products of sine and cosine functions.

$$\begin{aligned} 1. \quad f(n) &= \sin(a*n)*\sin(b*n) = \\ &\quad (1/2)*(\cos((a-b)*n)-\cos((a+b)*n)) \\ 2. \quad f(n) &= \cos(a*n)*\cos(b*n) = \\ &\quad (1/2)*(\cos((a-b)*n)+\cos((a+b)*n)) \\ 3. \quad f(n) &= \sin(a*n)*\cos(b*n) \\ &= (1/2)*(\sin((a+b)*n)+\sin((a-b)*n)) \end{aligned}$$

Sum and difference frequencies

For any pair of arbitrary values for **a** and **b**, the frequencies of the sinusoids in the resulting functions will be given by **a+b** and **a-b**.

*(These are often referred to as the sum and difference frequencies. Note that as **a** approaches **b**, the difference frequency approaches zero. This is what produces the constant values of 1/2 in Figure 9 and the positive bias on the black curve in Figures 10 and 11.)*

A form of measurement error

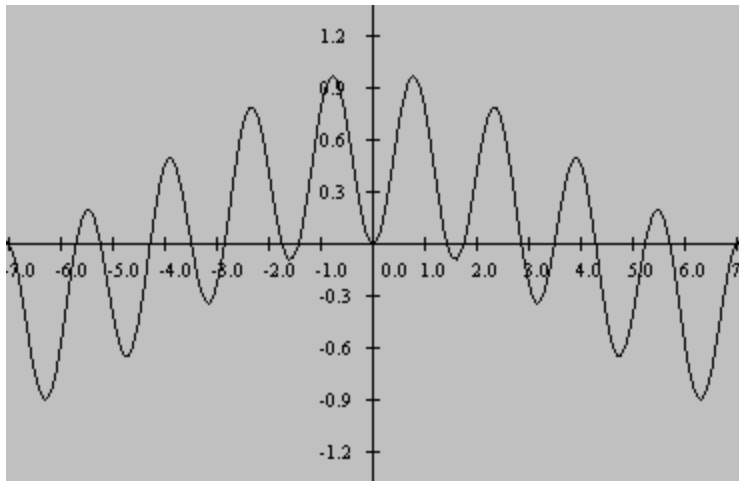
As a practical matter, if those resulting sum and difference frequencies are not multiples of one another, it will not be possible to perform the summation over an even number of cycles of both sinusoids. Therefore, one or the other, or perhaps both, will contribute a small amount to the sum of products due to including a partial cycle in the summation. This is a form of measurement error that occurs when performing frequency spectrum analysis using Fourier transform methods.

(The percentage contribution of this error to the average value decreases as the number of cycles included in the average increases.)

Product of two sine functions at different frequencies

Consider the product of two sine functions at different frequencies as shown in [Figure 14](#). This is a plot of the function produced by multiplying $\sin(1.8x)$ by $\sin(2.2x)$.

Figure 14. Plot of $\sin(1.8x) \cdot \sin(2.2x)$.



The high frequency component shown in [Figure 14](#) is the component attributable to $\mathbf{a+b}$. The long sweeping low frequency component is the component attributable to $\mathbf{a-b}$.

The sum of the product function

Judging from the graph in [Figure 14](#) , performing a summation of the product function from -7 to +7 would include almost exactly nine cycles of the high frequency component. Thus, the high frequency component would contribute very little, if anything, to the summation.

However, the summation would include less than one complete cycle of the low frequency component. Therefore, the low frequency component would contribute some output to the summation in the form of a measurement error.

Similar results occur when multiplying a cosine function by a cosine function having a different frequency, or when multiplying a cosine function by a sine function having a different frequency. Graphical results for these two cases are shown in [Figure 15](#) and [Figure 16](#) .

Figure 15. Plot of $\cos(1.8x) \cdot \cos(2.2x)$.

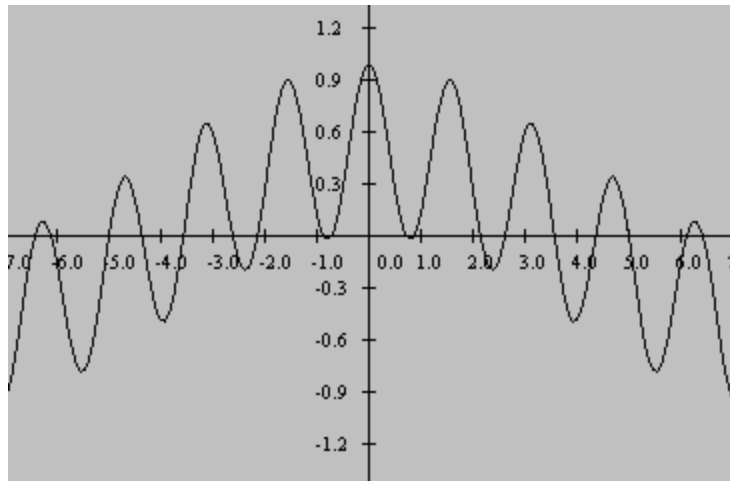
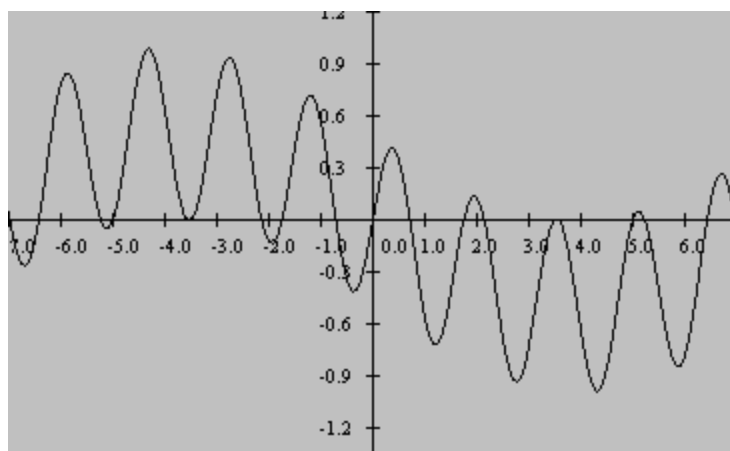


Figure 16. Plot of $\sin(1.8x) \cdot \cos(2.2x)$.



Once again, unless the summation interval includes an exact number of samples of both the sum frequency and the difference frequency, one of both of the sinusoids contained in the product function will contribute a measurement error to the result.

Summary

In this module, I have provided a quasi-theoretical basis for frequency spectrum analysis.

A pure theoretical basis for frequency spectrum analysis involves some rather complicated mathematics and is somewhat difficult to understand. However, from a practical viewpoint, it is not too difficult to understand how the complex mathematics produce the results that they produce.

Hopefully the quasi-theoretical explanations provided in this module will help you to understand what makes spectrum analysis work.

What's next?

The next module in this series will reduce much of what I have discussed in this module to practice. I will present and explain a program that implements a DFT algorithm for performing frequency spectrum analysis. In addition, I will present the results of several interesting experiments in frequency spectrum analysis using that algorithm.

A subsequent module will explain some of the signal processing concepts that made it possible for the inventors of the FFT algorithm to design a computational algorithm that is much faster than the DFT algorithm. As is often the case, however, the FFT algorithm trades off speed for generality.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1478-Fun with Java, How and Why Spectral Analysis Works
- File: Java1478.htm
- Published: 06/29/04

Baldwin explains how the Fourier transform can be used to determine the spectral content of a signal in the time domain.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1482-Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm

Baldwin explains several different programs used for spectral analysis including a DFT program and an FFT program. He also explains the impact of the sampling frequency and the Nyquist folding frequency on spectral analysis.

Revised: Fri Oct 16 23:17:43 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Sampling frequency and the Nyquist folding frequency](#)
 - [Display sinusoids](#)
 - [The plotting program named Graph06](#)
 - [The Nyquist folding frequency](#)
 - [One more example](#)
 - [Beginning of the class named Dsp029](#)
 - [The getNmbr method](#)
 - [The method named f1](#)

- [The program named Graph06](#)
- [Spectral Analysis using a DFT Algorithm](#)
 - [The spectral analysis output](#)
 - [Another DFT example](#)
 - [The program named Dsp028](#)
 - [Beginning of the class named Dsp028](#)
 - [The program named Graph03](#)
 - [The transform method of the ForwardRealToComplex01 class](#)
 - [The beginning of the transform method](#)
- [Spectral analysis using an FFT algorithm](#)
 - [The program named Dsp030](#)
 - [The ForwardRealToComplexFFT01 class](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

The how and the why of spectral analysis

A previous module titled [Fun with Java, How and Why Spectral Analysis Works](#) explained how and why spectral analysis works. An understanding of that module is a prerequisite to understanding this module.

Programs to perform spectral analysis

In this module I will provide and explain different programs used for performing spectral analysis. The first program is a very general program that implements a *Discrete Fourier Transform (DFT)* algorithm. I will explain this program in detail.

The second program is a less general, but much faster program that implements a *Fast Fourier Transform (FFT)* algorithm. I will defer an explanation of this program until a future module. I am providing it here so that you can use it and compare it with the DFT program in terms of speed and flexibility.

Fundamental aspects of spectral analysis

I will use the DFT program to illustrate several fundamental aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

Visual illustration of sampling

I will also provide and explain a program that produces a visual illustration of the impact of the sampling frequency and the Nyquist folding frequency.

Plotting programs

Finally, I will provide, but will not explain two different programs used for display purposes. These are newer versions of graphics display programs that I explained in the module titled [Plotting Engineering and Scientific Data using Java](#).

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Required parameters.
- [Figure 2.](#) Actual parameter values.
- [Figure 3.](#) Program output for five sinusoids.
- [Figure 4.](#) A new set of frequency values.

- [Figure 5.](#) Program output for five sinusoids.
- [Figure 6.](#) One more example.
- [Figure 7.](#) Program output for five sinusoids.
- [Figure 8.](#) Parameters for spectral analysis.
- [Figure 9.](#) Spectral analysis of five sinusoids.
- [Figure 10.](#) The input parameters.
- [Figure 11.](#) Spectral analysis of five sinusoids.
- [Figure 12.](#) Required input parameters for Dsp028.
- [Figure 13.](#) Contents of Dsp028.txt file.
- [Figure 14.](#) Output from Graph03.
- [Figure 15.](#) Spectral transform expressions.
- [Figure 16.](#) Required input parameters for Dsp030.
- [Figure 17.](#) Example input parameters.
- [Figure 18.](#) FFT of five sinusoids.
- [Figure 19.](#) Parameters for A matching DFT spectral analysis.
- [Figure 20.](#) DFT of five sinusoids.

Listings

- [Listing 1.](#) Beginning of the class named Dsp029.
- [Listing 2.](#) Create array objects to hold sinusoidal data.
- [Listing 3.](#) Get the parameters.
- [Listing 4.](#) Create the sinusoidal data.
- [Listing 5.](#) The code for GraphIntfc01.
- [Listing 6.](#) The getNmbr method.
- [Listing 7.](#) The method named f1.
- [Listing 8.](#) Beginning of the class named Dsp028.
- [Listing 9.](#) Declare array variables.
- [Listing 10.](#) Beginning of the constructor.
- [Listing 11.](#) Create the raw sinusoidal data.
- [Listing 12.](#) Perform the spectral analysis.
- [Listing 13.](#) The method named f1.
- [Listing 14.](#) The beginning of the transform method.
- [Listing 15.](#) The remainder of the method and the class.
- [Listing 16.](#) Dsp029.java.
- [Listing 17.](#) GraphIntfc01.java.
- [Listing 18.](#) Graph06.java.

- [Listing 19.](#) Dsp028.java.
- [Listing 20.](#) Graph03.java.
- [Listing 21.](#) ForwardRealToComplex01.java.
- [Listing 22.](#) Dsp030.java.
- [Listing 23.](#) ForwardRealToComplexFFT01.java.

Preview

Before I get into the technical details, here is a preview of the programs and their purposes that I will present and explain in this module:

- Dsp029 - Provides a visual illustration of the impact of the sampling frequency and the Nyquist folding frequency.
- Dsp028 - Driver program for doing spectral analysis using a DFT algorithm.
- ForwardRealToComplex01 - Class that implements the DFT algorithm.
- Dsp030 - Driver program for doing spectral analysis using an FFT algorithm.
- ForwardRealToComplexFFT01 - Class that implements the FFT algorithm (will defer explanation until a future module).
- Graph03 - Used to display results of spectral analysis. (*The concepts were explained in the earlier module titled [Plotting Engineering and Scientific Data using Java.](#)*)
- Graph06 - Used to display the impact of sampling frequency and the Nyquist folding frequency. Also used to display the results of spectral analysis. (*The concepts were explained in the earlier module titled [Plotting Engineering and Scientific Data using Java.](#)*)

Discussion and sample code

This will be a long module involving lots of code and lots of explanations, so fill your cup with java and let's get started.

Sampling frequency and the Nyquist folding frequency

I will begin the discussion with the program named **Dsp029** , which provides a visual illustration of the impact of the sampling frequency and the Nyquist folding frequency. A complete listing of this program is shown in [Listing 16](#) near the end of the module.

Display sinusoids

This program generates and displays up to five sinusoids having the same sampling frequency but having different sinusoidal frequencies and amplitudes. The program provides a visual illustration of the way in which frequencies above one-half the sampling frequency fold back into the area bounded by zero and one-half the sampling frequency.

(The frequency at one-half the sampling frequency is known as the Nyquist folding frequency.)

Input parameters

The program gets its input parameters from a file named **Dsp029.txt** . If that file doesn't exist in the current directory, the program uses a set of default parameters.

Each parameter value must be stored as characters on a separate line in the file named **Dsp029.txt** . The required parameters are shown in [Figure 1](#).

Figure 1. Required parameters.

Figure 1. Required parameters.

```
Data length as type int
Number of sinusoids as type int.  Max value is
5.
List of sinusoid frequency values as type
double.
List of sinusoid amplitude values as type
double.
```

The length of the two lists

The number of values in each of the lists must match the value for the number of sinusoids. Also, you must not allow blank lines at the end of the data in the file.

Frequency value specifications

Each frequency value is specified as a type double value representing a fractional part of the sampling frequency.

(For example, a double value of 0.5 specifies one-half the sampling frequency, or the Nyquist folding frequency. A double value of 2.0 specifies a frequency that is twice the sampling frequency.)

[Figure 2](#) shows the contents of the file named **Dsp029.txt** that I used to produce the output shown in [Figure 3](#). I will discuss that output later.

Figure 2. Actual parameter values.

```
50.0
5
0.03125
0.0625
0.125
0.25
0.5
90
90
90
90
90
```

The plotting program named Graph06

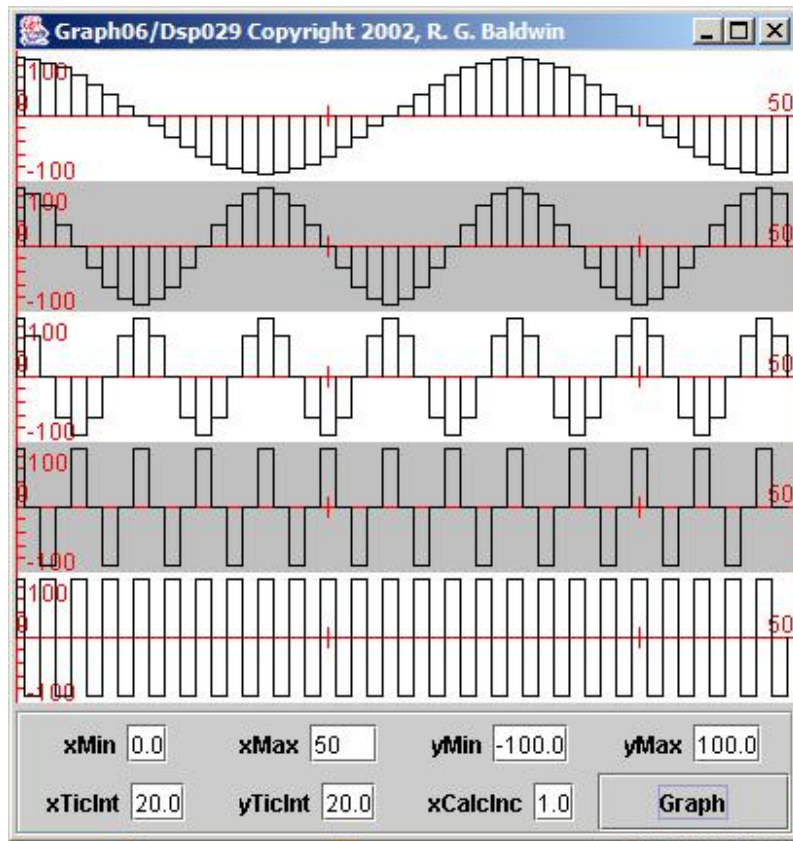
The plotting program that is used to plot the output data from this program requires that the program implement **GraphIntfc01** . I discussed that interface in the module titled [Plotting Engineering and Scientific Data using Java](#) . For example, the plotting program named **Graph06** can be used to plot the data produced by this program. When it is used, the program is executed and its output is plotted by entering the following at the command line prompt:

```
java Graph06 Dsp029
```

Program output

[Figure 3](#) shows the output produced by running the program named **Dsp029** with the parameters shown in [Figure 2](#) and then adjusting the xMax parameter in the textbox at the bottom of the display.

Figure 3. Program output for five sinusoids.



Five horizontal plots

Each of the five horizontal plots in [Figure 3](#) shows a sampled sinusoid. Each of the vertical bars represents one sample value for a given sinusoid.

If you examine the frequency values in [Figure 2](#) carefully, you will see that they represent the sampling frequency divided by the factors 32, 16, 8, 4, and 2. Thus, the last frequency value is the Nyquist folding frequency and the first four frequency values are related to that frequency by multiples of two.

The horizontal plot at the top of [Figure 3](#) is a reasonably well defined cosine wave. The horizontal plot at the bottom of [Figure 3](#) shows the result of having exactly two samples per cycle of the sinusoid. Using this plotting scheme, the

sampled sinusoid is represented as a square wave. Using another plotting scheme (such as that used in the program named **Graph03**) the sampled sinusoid at the bottom would be represented as a triangular wave.

An upper frequency limit

Regardless of the plotting scheme used, it should be obvious that a set of uniform samples cannot possibly represent frequencies higher than one-half the sampling frequency because then there would be less than one sample per cycle of the sinusoid.

The Nyquist folding frequency

Now I will show you why the frequency at one-half the sampling frequency is referred to as the folding frequency using the new set of frequency values shown in [Figure 4](#).

([Figure 4](#) shows the values read from the file named **Dsp029.txt** and displayed in an improved format.)

In this case, the third frequency in the list is one-half the sampling frequency, which is the folding frequency. The two frequencies on either side of that one have values that are symmetrical about the folding frequency.

Figure 4. A new set of frequency values.

Figure 4. A new set of frequency values.

```
Data length: 50
Number sinusoids: 5
Frequencies
0.125
0.25
0.5
0.75
0.875
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

The program output

The five horizontal plots in [Figure 5](#) show the result of running the program named **Dsp029** with the frequencies shown in [Figure 4](#).

Figure 5. Program output for five sinusoids.

The screenshot displays the Graph06/Dsp029 software interface. The title bar reads "Graph06/Dsp029 Copyright 2002, R. G. Baldwin". The main window contains five vertically stacked bar graphs. The top graph has a white background, while the others have a gray background. Each graph has a vertical axis ranging from -100 to 100, with major ticks at -100, -50, 0, 50, and 100. The horizontal axis represents a range from 0.0 to 50.0, with major ticks at 0.0, 20.0, and 50.0. The bars are black outlines, and the area between the bars is filled with the background color. The control panel at the bottom includes the following fields and buttons:

- xMin**: 0.0
- xMax**: 50
- yMin**: -100.0
- yMax**: 100.0
- xTicInt**: 20.0
- yTicInt**: 20.0
- xCalcInc**: 1.0
- Graph** button

The top two plots in [Figure 5](#) are obviously the sampled representations of the two lower frequencies specified by the first two frequencies in [Figure 4](#).

However, it is not so obvious that the bottom two plots in [Figure 5](#) are the sampled representations of the two higher frequencies specified by the last two frequencies in [Figure 4](#). They look exactly like the top two plots but in reverse order.

Unable to distinguish ...

Frequencies above one-half the sampling frequency are not distinguishable by viewing the sampled data. In fact, they are converted to lower frequencies by sampling process. The new lower frequencies fold around a point in the frequency spectrum given by one-half the sampling frequency. That is why it is called the folding frequency. (In 1933, this frequency was named after scientist [Harry Nyquist](#).)

One more example

Let's look at one more example of plotted sinusoids. Consider the frequency values shown in [Figure 6](#). The second and third frequencies are symmetrical about the sampling frequency. The fourth and fifth frequencies are symmetrical about twice the sampling frequency. The first frequency value is the same distance from zero as the other four frequencies are from the sampling frequency and twice the sampling frequency.

Figure 6. One more example.

Figure 6. One more example.

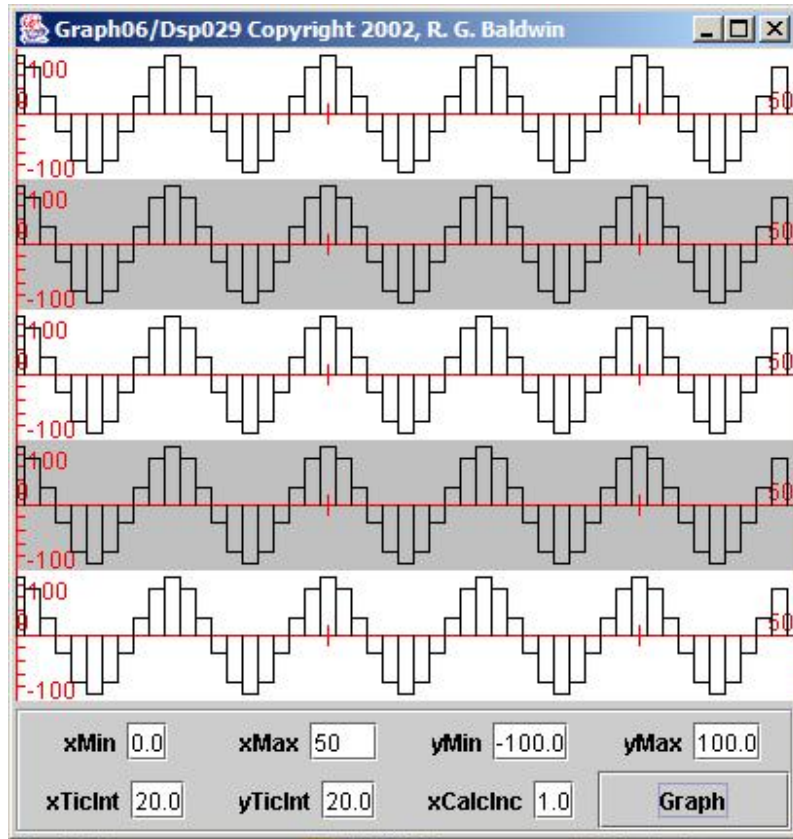
```
Data length: 50
Number sinusoids: 5
Frequencies
0.1
0.9
1.1
1.9
2.1
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

The program output

[Figure 7](#) shows the output produced by running the program named **Dsp029** with the frequency parameters specified by [Figure 6](#).

Figure 7. Program output for five sinusoids.

Figure 7. Program output for five sinusoids.



The sinusoids are indistinguishable

Although the actual frequencies of the five cosine functions are significantly different, once they are sampled, they are indistinguishable. The sampling process converts the actual frequencies to new frequencies that not only fold around one-half the sampling frequency, they also fold around all multiples of one-half the sampling frequency.

Beginning of the class named Dsp029

The program named **Dsp029** is provided in [Listing 16](#) near the end of the module. I will break the program down and explain it in fragments, beginning with the fragment shown in [Listing 1](#).

Listing 1. Beginning of the class named Dsp029.

```
class Dsp029 implements GraphIntf01{
    final double pi = Math.PI;//for simplification

    //Begin default parameters
    int len = 400;//data length
    int numberSinusoids = 5;
    //Frequencies of the sinusoids
    double[] freq = {0.1,0.25,0.5,0.75,0.9};
    //Amplitudes of the sinusoids
    double[] amp = {75,75,75,75,75};
    //End default parameters
```

The code in [Listing 1](#) defines a convenience constant representing the value of **pi** and then defines the set of default parameters that will be used by the program in the event that the file named **Dsp029.txt** does not exist in the current directory.

Create array objects to hold sinusoidal data

The code in [Listing 2](#) creates five array objects that will be populated with sinusoidal data.

Listing 2. Create array objects to hold sinusoidal data.

```
double[] data1 = new double[len];  
double[] data2 = new double[len];  
double[] data3 = new double[len];  
double[] data4 = new double[len];  
double[] data5 = new double[len];
```

Get the parameters

The constructor begins in [Listing 3](#). The code in this fragment calls the method named **getParameters** to read the parameters from the file named **Dsp029.txt**.

Listing 3. Get the parameters.

```
public Dsp029(){//constructor  
  
    if(new File("Dsp029.txt").exists()){  
        getParameters();  
    }//end if
```

Before calling the **getParameters** method, however, the program calls the **exists** method of the **File** class to confirm that the file actually exists. If the file doesn't exist, the call to **getParameters** is skipped, causing the default parameters defined in [Listing 1](#) to be used instead.

The `getParameters` method

The **`getParameters`** method is straightforward, so I won't discuss it in detail. You can view it in [Listing 16](#). Suffice it to say that the method reads the input parameters from the disk file and writes their values into the variables declared in [Listing 1](#), overwriting the default values stored in those variables.

In addition, the **`getParameters`** method displays the values read from the disk file in the format shown in [Figure 4](#) and [Figure 6](#).

Create the sinusoidal data

For simplicity, this program always generates five sinusoids, even if fewer than five were requested as the input parameter value for **`numberSinusoids`**. In that case, the extra sinusoids are generated using default values and are simply ignored when the sinusoids are plotted.

The code fragment in [Listing 4](#) creates the sinusoidal data for each of the five specified frequencies and saves that data in the array objects that were created in [Listing 2](#).

Listing 4. Create the sinusoidal data.

Listing 4. Create the sinusoidal data.

```
        for(int n = 0;n < len;n++){
            data1[n] =
amp[0]*Math.cos(2*pi*n*freq[0]);
            data2[n] =
amp[1]*Math.cos(2*pi*n*freq[1]);
            data3[n] =
amp[2]*Math.cos(2*pi*n*freq[2]);
            data4[n] =
amp[3]*Math.cos(2*pi*n*freq[3]);
            data5[n] =
amp[4]*Math.cos(2*pi*n*freq[4]);
        }//end for loop

    }//end constructor
```

The end of the constructor

[Listing 4](#) also signals the end of the constructor. When the constructor terminates, an object of the **Dsp029** class has been instantiated. The five arrays shown in [Listing 4](#) have been populated with sinusoidal data according to the parameters read from the file named **Dsp029.txt** or according to the default values of the parameters shown in [Listing 1](#).

Plotting the sinusoidal data

In order to better understand what is going on in the plotting process, it would be helpful for you to review the module titled [Plotting Engineering and Scientific Data using Java](#). However, assuming that you don't have the time to do that, I will provide a very brief explanation as to how the plotting programs work.

Using Graph06 to plot the sinusoidal data

The plotting program named **Graph06** can be used to plot the sinusoidal data as follows:

- Define and compile the program named **Dsp029** , implementing the interface named **GraphIntfc01** . The definition of that interface is shown in [Listing 5](#)
- Compile the program named **Graph06** . Then start the plotting program named **Graph06** running by entering the command shown below at the command line prompt.

```
java Graph06 Dsp029
```

The code for GraphIntfc01

Listing 5. The code for GraphIntfc01.

```
public interface GraphIntfc01{
    public int getNmbr();
    public double f1(double x);
    public double f2(double x);
    public double f3(double x);
    public double f4(double x);
    public double f5(double x);
} //end GraphIntfc01
```

What does this do?

When executed in this manner, the program named **Graph06** instantiates an object of the class named **Dsp029** and then calls the interface methods on that object to obtain the data to be plotted.

In this case, the constructor for the **Dsp029** class populates the five array objects with sinusoidal data. The subsequent call to the interface methods by the program named **Graph06** causes that sinusoidal data to be retrieved and plotted by Graph06.

The GraphIntfc01 interface methods

A brief description of each of the interface methods is provided in the following sections.

The getNnbr method

Plotting programs based on **GraphIntfc01** can be used to plot any number of functions from one to five.

The method named **getNnbr** must return an integer value between 1 and 5 that specifies the number of functions to be plotted. The plotting program uses that value to divide the total plotting surface into the specified number of plotting areas, and plots each of the functions named **f1** through **fn** in one of those plotting areas.

The methods named f1, f2, f3, f4, and f5

As you can see in [Listing 5](#), each of these methods receives a double value as an incoming parameter and returns a double value. In essence, each of these methods receives a value for the horizontal coordinate x and returns the corresponding value for the vertical coordinate y.

One plotting area per method

Each of these methods provides the data to be plotted in one plotting area. The method named **f1** provides the data for the top plotting area; the method named **f2** provides the data for the first plotting area down from the top, and so forth.

(For example, if the `getNmbr` method returns a value of 4, the method named `f5` will never be called. If `getNmbr` returns 5, the method named `f5` will be called to provide the data for the bottom plotting area.)

How does it work?

Each plotting area contains a horizontal axis. The plotting program moves across the horizontal axis in each plotting area one step at a time (*moving in incremental steps equal to the plotting parameter named `xCalcInc`, which you will find if you examine the code for **Graph06***).

At each step along the way, the plotting program calls the method associated with that plotting area, (***f1** , **f2** , etc.*) , passing the horizontal position as a parameter to the method.

The value returned by the method is assumed to be the vertical value associated with that horizontal position, and that is the vertical value that is plotted for that horizontal position.

Doesn't know and doesn't care

The plotting program doesn't know, and doesn't care how the interface method decides on the value to return for each value that it receives as an incoming parameter. The plotting program simply calls the methods to get the data, and then plots the returned values.

Computed "on the fly"

For example, the returned values could be computed and returned "on the fly," as was the case in the example program named **Graph01Demo** , which I explained in the module titled [Plotting Engineering and Scientific Data using Java](#).

Returned from an array

On the other hand, the values could have been computed earlier and saved in an array. That is the case with all the programs that I will explain in this module.

From a disk file, a database, the Internet, etc.

The returned values could be read from a disk file, obtained from a database on another computer, or obtained from any other source such as another computer on the Internet.

All that matters is that when the plotting program calls one of the five methods named **f1** through **f5** , passing a **double** value as a parameter, it expects to receive a **double** value in return, and it will plot the value that it receives.

The `getNmbr` method

The **getNmbr** method for the class named **Dsp029** is shown in [Listing 6](#).

Listing 6. The `getNmbr` method.

Listing 6. The getNmbr method.

```
public int getNmbr(){
    //Return number of functions to
    // process. Must not exceed 5.
    return numberSinusoids;
} //end getNmbr
```

This is a very simple method. It returns the value stored in the variable named **numberSinusoids** . This variable may contain the default value established by [Listing 1](#), or may contain the value read from the file named **Dsp029.txt** by the method named **getParameters** .

The method named f1

The code for the method named **f1** is shown in [Listing 7](#).

Listing 7. The method named f1.

```
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data1.length-1){
        return 0;
    }else{
        return data1[index];
    } //end else
} //end function
```

Note that there is not a one-to-one correspondence between horizontal coordinate values and pixels on the screen. For example, it may sometimes be necessary to plot 90 values across an area of the screen containing 110 pixels. The plotting program must interpolate properly to deal with that issue. Therefore, the plotting program deals with horizontal coordinates as type **double** and then converts those coordinate values to integer pixel values when the time comes to actually draw the material on the screen.

Round from double to int

The method named **f1** receives an incoming horizontal coordinate value as type **double** . It rounds that value to the nearest value of type **long** and casts it to type **int** to determine the index value to use when retrieving the corresponding vertical value from the array object.

If the index value is outside the bounds of the array, the method simply returns a value of zero. Otherwise, it uses the index value to return the value stored in the array object at that index.

The remaining interface methods

The remaining four interface methods are identical to the method named **f1** , except that each method returns data values stored in a different array object. Therefore, I won't discuss those methods.

That's it for Dsp029

That's about it for the program named **Dsp029** . If you understand this program, you are well ahead of the game. The overall structure for the programs named **Dsp028** and **Dsp030** are very similar to the structure for **Dsp029** . The big difference is the manner in which they populate the array objects with the data that is to be plotted. Instead of simply plotting sinusoids, they perform spectral analysis on sinusoids and provide the results of the spectral analysis to be plotted.

Using the interface named GraphIntfc01

As you learned earlier, this is a very simple interface. However, because the class named **Dsp029** implements the interface, the interface definition file must be in the same directory as the source file for **Dsp029** in order to successfully compile **Dsp029**. Therefore, I have provided a complete listing of **GraphIntfc01** in [Listing 17](#) near the end of the module.

The program named Graph06

A complete listing of the program named **Graph06** is provided in [Listing 18](#) near the end of the module. This is simply a newer version of graphics display programs that I explained in the earlier module titled [Plotting Engineering and Scientific Data using Java](#). Therefore, I won't repeat that explanation here. The comments at the beginning and spread throughout the program provide considerable information about it.

Operational aspects of Graph06

However, an explanation of the operational aspects of the program will be useful here. You can use this program to display the output produced by **Dsp029** by entering the following at the command line prompt:

```
java Graph06 Dsp029
```

As you saw in [Figure 3](#) and other previous figures, this program provides the following text fields for user input, along with a button labeled **Graph** :

- xMin = minimum x-axis value
- xMax = maximum x-axis value
- yMin = minimum y-axis value
- yMax = maximum y-axis value
- xTicInt = tic interval on x-axis
- yTicInt = tic interval on y-axis
- xCalcInc = calculation interval

These text fields make it possible for you to adjust the plotting parameters and to re-plot the graphs as many times as needed.

You can modify any of these parameters and then click the **Graph** button to cause the five functions to be re-plotted according to the new plotting parameters.

Spectral Analysis using a DFT Algorithm

Now that you have a good idea where we are heading, it's time to start doing some spectral analysis.

Let's begin by looking at some output obtained by performing a spectral analysis on the same five sinusoids shown in [Figure 3](#). The parameters used to perform this spectral analysis are shown in [Figure 8](#). (*I will explain each of these parameters as we go along.*)

Figure 8. Parameters for spectral analysis.

Figure 8. Parameters for spectral analysis.

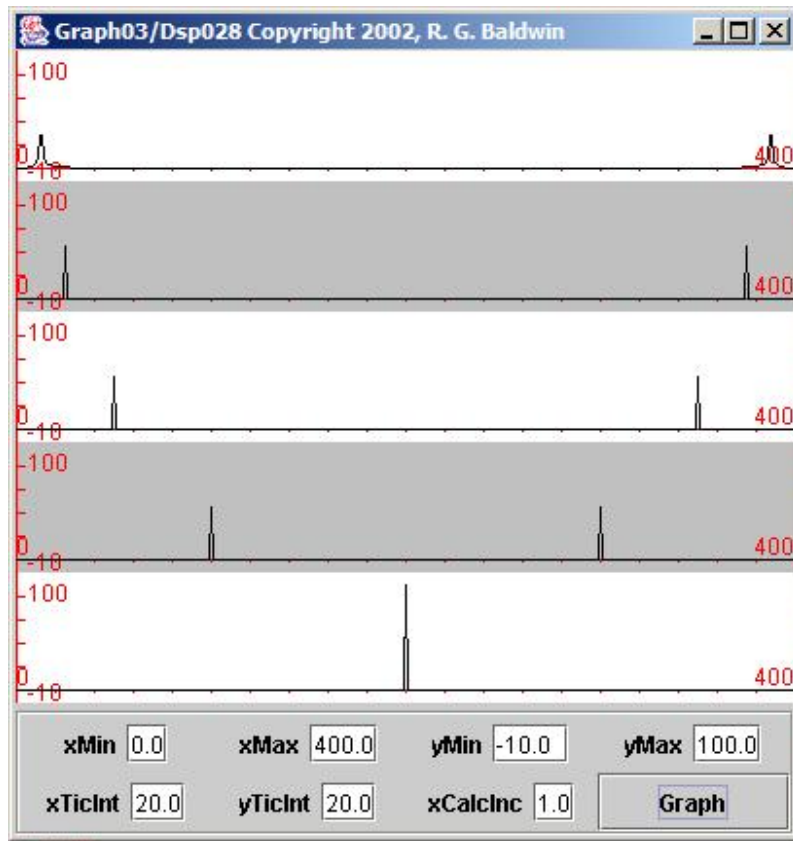
```
Data length: 400
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 1.0
Number spectra: 5
Frequencies
0.03125
0.0625
0.125
0.25
0.5
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

Although more parameters are required to perform spectral analysis than are required to simply generate and plot the sinusoids, the number of sinusoids, the frequencies of the sinusoids, and the amplitudes of the sinusoids in [Figure 8](#) are the same as in [Figure 2](#).

The spectral analysis output

The output produced by performing spectral analysis on these five sinusoids is shown in [Figure 9](#). The computation and display of this spectral analysis was performed using the programs named **Dsp028** and **Graph03**. (Note that the parameters in the text boxes at the bottom were used to alter the appearance of the plots.)

Figure 9. Spectral analysis of five sinusoids.



The format explained

Five separate spectral analyses were performed and the results of those five spectral analyses are shown in [Figure 9](#). Each of the horizontal lines in [Figure 9](#) is the horizontal axis used to display the result of performing a spectral analysis on a different sinusoid. In other words, [Figure 9](#) contains five separate graphs moving from the top to the bottom of the display. The individual graphs have alternating white and gray backgrounds to make them easier to separate visually.

The top graph in [Figure 9](#) shows the result of performing a spectral analysis on the top sinusoid in [Figure 3](#). Moving down the page, each graph in [Figure](#)

[9](#) shows the result of performing a spectral analysis on the corresponding sinusoid in [Figure 3](#).

The frequency axes

The horizontal axes in [Figure 9](#) represent the frequency range from zero to the sampling frequency.

(The frequency range covered is specified by the Lower frequency bound and the Upper frequency bound parameters in [Figure 8](#).)

The horizontal units

The horizontal units in [Figure 9](#) don't represent frequency in an absolute sense of cycles per second or Hertz. Rather, the horizontal units in [Figure 9](#) represent the frequency bins for which spectral energy was computed. In this case, the spectral energy for each sinusoid was computed in 400 equally spaced bins distributed between zero and the sampling frequency.

*(The number of frequency bins for each individual spectrum computed by this program is equal to the **Data** length parameter in [Figure 8](#). Those frequency bins are distributed uniformly between the **Lower** frequency bound and the **Upper** frequency bound parameters in [Figure 8](#).)*

Location of the folding frequency

Because the right-most end of each horizontal axis in [Figure 9](#) represents the sampling frequency, the center of each horizontal axis represents one-half the sampling frequency, or the Nyquist folding frequency. Thus, the frequency

represented by the center of each horizontal axis represents the frequency specified by a value of 0.5 in [Figure 8](#).

A peak at the folding frequency

You can see a large peak in energy at the folding frequency of the bottom graph in [Figure 9](#). That peak corresponds to the frequency of the fifth sinusoid specified in the parameters shown in [Figure 8](#). *(This also corresponds to the spectrum of the bottom graph in [Figure 3](#).)*

Knowing that, you should be able to correlate each of the peaks to the left of center in [Figure 9](#) with the frequencies of the sinusoids specified in [Figure 8](#) and with the individual sinusoids plotted in [Figure 3](#).

The frequency folding effect

[Figure 9](#) clearly shows the frequency folding effect of the sampling process illustrated earlier. As you can see, the peaks in the various graphs to the right of the folding frequency are mirror images of the peaks to the left of the folding frequency. In other words, given a set of samples of a sinusoid, the spectral analysis process is unable to determine whether the peak is above or below the folding frequency, so the energy is equally distributed between two peaks on opposite sides of the folding frequency.

Size of the peak at the folding frequency

Note that the peak in the bottom graph is approximately twice the height of the peaks in the other graphs. This is because the peak at the folding frequency has no mirror-image partner, and all the energy is concentrated in that single peak.

(Another interpretation is that two mirror-image peaks converge at the folding frequency causing the resulting peak to be twice as large

as either mirror-image peak. I will illustrate this effect with another example later.)

A short fat peak at the top

You may also have noticed that the peaks in the top graph are shorter and wider than the peaks in the other graphs. This may be because the actual frequency of the sinusoid for the top graph is about half way between the values of the twelfth and thirteenth bins for which spectral energy was computed. Thus, the energy in the sinusoid was spread between the bins on either side of the actual frequency.

This frequency spreading effect can be minimized by increasing the data length to 800 samples. This causes the frequency bins to be only half as wide and the peak in the top graph becomes tall and narrow just like the peaks in the other graphs. You should try this and observe the result when you run the program later.

It is also instructive to plot these spectra with a data length of 400 using the program named **Graph06** . This will show you how the energy is distributed between the frequency bins. This is most effective when the graph is expanded as described in the next section.

Mapping the peaks to pixels

The broadening of the peak in the top graph may also have to do with the requirement to map the peaks in the spectrum to the locations of the actual pixels on the screen. If the location of the peak falls between the positions of two pixels, the plotting program must interpolate the energy in the peak so as to display that energy in actual pixel locations.

This effect can be minimized by plotting the same number of spectral values across a wider area of the screen. When you run this program later, click the maximize button on the **Frame** to cause the display to occupy the entire screen. That will give you a much better look at the actual shape of each of the peaks. Do this using both **Graph03** and **Graph06** to plot the results.

(Note: When switching between the plotting programs, you may need to delete the class files from the old program and compile the new program to avoid having class files with the same names from the two programs becoming intermingled in the same directory.)

Another DFT example

This next example is designed to illustrate the following features of the DFT algorithm which don't generally apply to an FFT algorithm:

- Ability to do spectral analysis on data of arbitrary lengths. *(With many FFT algorithms, the data length must be a power of two.)*
- Ability to zero in on an arbitrary range of frequencies and to ignore all other frequencies. *(Most FFT algorithms always compute the spectrum at uniform frequency increments from zero to one unit less than the sampling frequency.)*

As mentioned earlier, the DFT algorithm is much more flexible while the FFT algorithm is much faster, particularly for large data lengths.

Peaks merge at the folding frequency

In addition to illustrating these fundamental aspects of the DFT algorithm, this example also illustrates how the mirror image peaks on either side of the folding frequency merge into a single larger peak as the data frequency approaches the folding frequency.

The input parameters

The input parameters are shown in [Figure 10](#). Note in particular the values for the following parameters:

- Data length: 200

- Sample for zero time: 0
- Lower frequency bound: 0.4
- Upper frequency bound: 0.6

Figure 10. The input parameters.

```
Data length: 200
Sample for zero time: 0
Lower frequency bound: 0.4
Upper frequency bound: 0.6
Number spectra: 5
Frequencies
0.492
0.494
0.496
0.498
0.5
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

A different data length

As you can see, the data length for this experiment is different from the data length of 400 used earlier. In addition, neither data length is a power of two.

Computational frequency bounds

As you can also see, the lower and upper frequency bounds are not 0.0 and 1.0 as in the earlier cases. In this case, the frequency bounds describe a much narrower range centered on the folding frequency.

Frequencies are close to the folding frequency

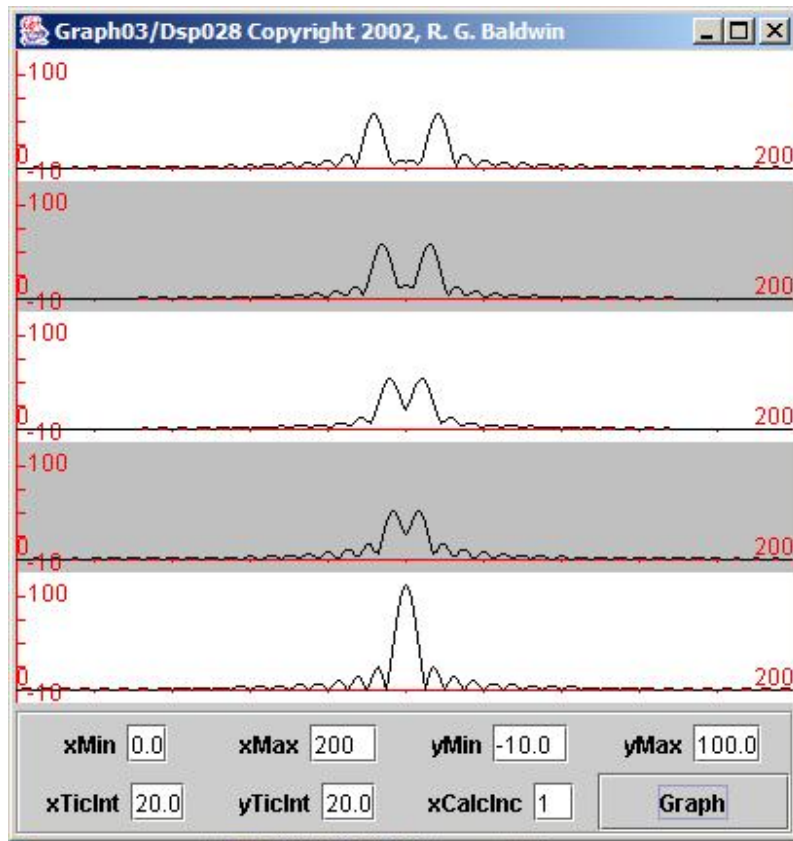
Finally, the frequencies of each of the five sinusoids specified in [Figure 10](#) is progressively closer to the folding frequency with the frequency of the fifth sinusoid being equal to the folding frequency.

The spectral analysis output

The output from the spectral analysis for each of the five sinusoids is shown in [Figure 11](#). *(Another interesting view of the same results is shown later in [Figure 14](#).)*

Figure 11. Spectral analysis of five sinusoids.

Figure 11. Spectral analysis of five sinusoids.



Peaks are symmetrical about the folding frequency

The spectral peaks shown in [Figure 11](#) are symmetrical about the folding frequency, which in turn is centered horizontally in each of the graphs. As you already know, the peaks are always symmetrical about the folding frequency due to the frequency folding at that frequency.

The folding frequency is centered horizontally due to the way that I defined the lower and upper frequency bounds, and the way that I adjusted the plotting parameters.

Peaks are well defined and wider than before

The peaks are well defined because I computed the spectral energy at 200 points across the specified frequency range from 0.4 to 0.6. Thus, the frequency bins at which I computed spectral energy were much narrower than before.

The peaks are wider because I displayed a much smaller slice of the entire frequency spectrum in the same physical screen space.

Peaks merge at the folding frequency

As the frequency of each sinusoid approaches the folding frequency, the two mirror-image peaks corresponding to that sinusoid merge into a single peak with twice the height at the folding frequency. This agrees with what you saw in [Figure 9](#), but on a much more detailed basis.

It would be difficult to perform this experiment using an FFT algorithm because of the inherent limitations built into the algorithm. The FFT algorithm sacrifices flexibility for speed.

Implementing the DFT algorithm

At this point, I will present and explain two different programs:

- Dsp028 - Driver program for doing spectral analysis using a DFT algorithm.
- ForwardRealToComplex01 - Class that implements the DFT algorithm.

In addition, I will present, but will not explain the plotting program named **Graph03**.

The program named Dsp028

This driver program is similar in many respects to the program named **Dsp029** that I explained earlier. It differs mainly in how it populates the array

objects containing the data that is plotted by the plotting program.

The program named **Dsp029** simply populates those array objects with five sinusoidal functions. The program named **Dsp028** also creates five sinusoidal functions. However, it passes those functions to a static method named **transform** belonging to the **ForwardRealToComplex01** class to perform spectral analysis on those functions. The results of the spectral analysis are used to populate the five array objects whose contents are plotted by the plotting program.

Because of the similarity of the two programs, my discussion of **Dsp028** will be much more brief than was my discussion of **Dsp029** .

Computes and displays the magnitude spectrum

The program named **Dsp028** computes and displays the magnitude of the spectral content for up to five sinusoids having different frequencies and amplitudes.

(Future modules will discuss other aspects of spectral analysis such as the complex spectrum, zero time, and the phase angle.)

Input parameters

The program gets input parameters from a file named **Dsp028.txt** . If that file doesn't exist in the current directory, the program uses a set of default parameters. As with the program named **Dsp029** , each parameter value must be stored as characters on a separate line in the file named **Dsp028.txt** . The required parameters are shown in [Figure 12](#).

Figure 12. Required input parameters for Dsp028.

```
Data length as type int
Sample number representing zero time as type int
Lower frequency bound as type double
Upper frequency bound as type double
Number of spectra as type int.  Max value is 5.
List of sinusoid frequency values as type
double.
List of sinusoid amplitude values as type
double.
```

Don't allow blank lines at the end of the data in the file.

The number of values in each of the lists must match the value for the number of spectra.

Specification of sinusoidal frequencies

As before, each frequency value is specified as a **double** value representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is one-half the sampling frequency.

Example contents for Dsp028.txt

[Figure 13](#) shows the contents of the file named **Dsp028.txt** that represent the parameters shown in [Figure 10](#).

Figure 13. Contents of Dsp028.txt file.

```
200
0
0.4
0.6
5
0.492
0.494
0.496
0.498
0.5
90
90
90
90
90
```

Performing the spectral analysis

A static method named **transform** belonging to the class named **ForwardRealToComplex01** is used to perform the actual spectral analysis. The method named **transform** does not implement an FFT algorithm. Rather, it is more general than, but much slower than an FFT algorithm.

Will discuss the code in fragments

As usual, I will discuss the code in fragments. A complete listing of the program is presented in [Listing 19](#) near the end of the module. Because of the similarity of **Dsp028** with **Dsp029** discussed earlier, the fragments for **Dsp028** will be much larger and will be explained in much less detail.

Beginning of the class named Dsp028

The class definition begins in [Listing 8](#). For reasons that you already understand, this class implements the interface named GraphIntfc01.

Listing 8. Beginning of the class named Dsp028.

```
class Dsp028 implements GraphIntfc01{
    final double pi = Math.PI;//for simplification

    //Begin default parameters
    int len = 400;//data length
    //Sample that represents zero time.
    int zeroTime = 0;
    //Low and high frequency limits for the
    // spectral analysis.
    double lowF = 0.0;
    double highF = 1.0;
    int numberSpectra = 5;
    //Frequencies of the sinusoids
    double[] freq = {0.1,0.2,0.3,0.4,0.5};
    //Amplitudes of the sinusoids
    double[] amp = {60,70,80,90,100};
    //End default parameters
```

The code in [Listing 8](#) defines a set of default parameter values that are used in the event that a file named **Dsp028.txt** does not exist in the current directory.

Declare array variables

The code in [Listing 9](#) declares several array variables that will be used to point to array objects whose purposes are explained in the comments.

Listing 9. Declare array variables.

```
//Following arrays will contain data that is
// input to the spectral analysis process.
double[] data1;
double[] data2;
double[] data3;
double[] data4;
double[] data5;

//Following arrays receive information back
// from the spectral analysis that is not used
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] magnitude1;
double[] magnitude2;
double[] magnitude3;
double[] magnitude4;
double[] magnitude5;
```

The constructor

The constructor for the class begins in [Listing 10](#). The constructor begins by getting the parameters from a file named **Dsp028.txt** . If that file doesn't exist in the current directory, default parameters are used.

Listing 10. Beginning of the constructor.

```
public Dsp028(){//constructor

    if(new File("Dsp028.txt").exists()){
        getParameters();
    }//end if
```

Always processes five sinusoids

For simplicity, this program always processes five sinusoids, even if fewer than five were requested as the input parameter for **numberSpectra** . In that case, the extra sinusoids are processed using default values and simply ignored when the results are plotted.

Create the raw sinusoidal data

The code in [Listing 11](#) instantiates array objects and creates the sinusoidal data upon which spectral analysis will be performed.

Listing 11. Create the raw sinusoidal data.

```
//First create empty array objects.
double[] data1 = new double[len];
double[] data2 = new double[len];
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];
//Now populate the array objects
for(int n = 0;n < len;n++){
    data1[n] =
amp[0]*Math.cos(2*pi*n*freq[0]);
    data2[n] =
amp[1]*Math.cos(2*pi*n*freq[1]);
    data3[n] =
amp[2]*Math.cos(2*pi*n*freq[2]);
    data4[n] =
amp[3]*Math.cos(2*pi*n*freq[3]);
    data5[n] =
amp[4]*Math.cos(2*pi*n*freq[4]);
} //end for loop
```

Perform the spectral analysis

The code in [Listing 12](#) creates array objects to receive the results and calls the static **transform** method of the **forwardRealToComplex01** class five times in succession to perform the spectral analysis on each of the five sinusoids.

(I will explain the transform method that performs the spectral analysis shortly.)

Only the magnitude data is displayed by this program. Therefore, the arrays that receive the other spectral analysis results from the transform method are discarded each time a new spectral analysis is performed.

Listing 12. Perform the spectral analysis.

```
    magnitude1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1, real,
imag, angle, magnitude1, zeroTime, lowF, highF);

    magnitude2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2, real,
imag, angle, magnitude2, zeroTime, lowF, highF);

    magnitude3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3, real,
imag, angle, magnitude3, zeroTime, lowF, highF);

    magnitude4 = new double[len];
    real = new double[len];
```

Listing 12. Perform the spectral analysis.

```
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data4, real,
    imag, angle, magnitude4, zeroTime, lowF, highF);

    magnitude5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5, real,
    imag, angle, magnitude5, zeroTime, lowF, highF);
} //end constructor
```

The spectral magnitude results

Note that the magnitude results are saved in the array objects referred to by **magnitude1** , **magnitude2** , etc. This will be important later when I discuss the interface methods defined by **Dsp028** .

The end of the constructor

[Listing 12](#) also signals the end of the constructor. When the constructor terminates, the object has been instantiated and populated with spectral analysis results for five sinusoids using the parameters specified by the file named **Dsp028.txt**.

The `getParameters` method

The `getParameters` method used in this program is the same as that used in **Dsp029** , so I won't discuss it further.

The interface methods

The **Dsp028** class must define the same six interface methods as the **Dsp029** class, which I discussed earlier. The only difference in the interface methods is the identification of the array objects from which the methods return data when the methods are called.

The code in [Listing 13](#) is typical of the code for methods **f1** through **f5** . As you can see, these methods return the data stored in the magnitude arrays. These are the spectral analysis results that are plotted in [Figure 9](#), [Figure 11](#), and later in [Figure 14](#) .

Listing 13. The method named f1.

```
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude1.length-1){
        return 0;
    }else{
        return magnitude1[index];
    }//end else
}//end function
```

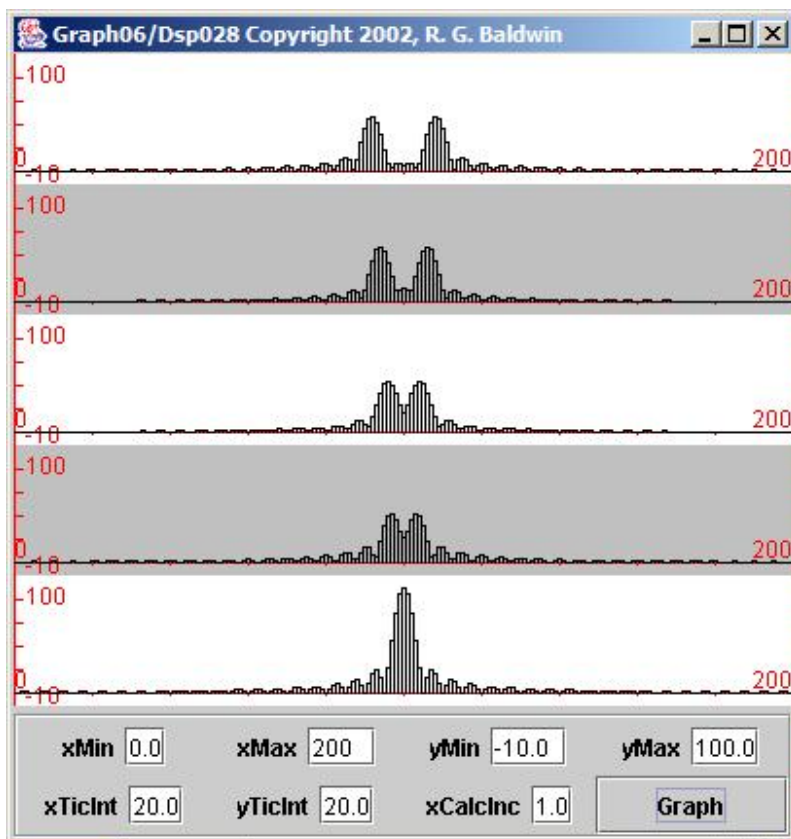
The program named Graph03

The plots in [Figure 9](#) and [Figure 11](#) were produced by entering the following at the command line prompt:

```
java Graph03 Dsp028
```

The program named **Graph03** is very similar to the program named **Graph06** discussed earlier. In fact, the program named **Graph06** can be used to produce very similar plots where the sample values are represented by vertical bars instead of being represented by connected dots. This results in the very interesting display shown in [Figure 14](#). Each of the vertical bars in [Figure 14](#) represents a computational frequency bin. (Compare [Figure 14](#) with [Figure 11](#).)

Figure 14. Output from Graph03.



In any event, **Graph03** is so similar to **Graph06** that I'm not going to discuss it further. A complete listing of the program named **Graph03** is provided in [Listing 20](#) near the end of the module.

The transform method of the ForwardRealToComplex01 class

That brings us to the heart of this module, which is the method that actually implements the DFT algorithm and performs the spectral analysis. This is a method named **transform**, which is a static method of the class named **ForwardRealToComplex01**. You saw this method being called five times in the code in [Listing 12](#).

Will discuss in fragments

As usual, I will discuss this method in fragments. A complete listing of the class is presented in [Listing 21](#) near the end of the module.

The **transform** method is a rather straightforward implementation of the concepts that I explained in the earlier module titled [Fun with Java, How and Why Spectral Analysis Works](#). If you have not done so already, I strongly urge you go to back and study that module at this time. You need to understand those concepts in order for the code in the **transform** method to make sense.

A brief description

For those of you who don't have the time to go back and study that module in detail, a brief description of the DFT algorithm follows.

Using a notation that I described in the earlier module, the expressions that you must evaluate to determine the frequency spectral content of a target time series at a frequency F are shown in [Figure 15](#).

Figure 15. Spectral transform expressions.

$$\begin{aligned}\text{Real}(F) &= \sum_{n=0}^{N-1} [x(n) * \cos(2\pi F * n)] \\ \text{Imag}(F) &= \sum_{n=0}^{N-1} [x(n) * \sin(2\pi F * n)] \\ \text{ComplexAmplitude}(F) &= \text{Real}(F) - j * \text{Imag}(F) \\ \text{Power}(F) &= \text{Real}(F) * \text{Real}(F) + \text{Imag}(F) * \text{Imag}(F) \\ \text{Amplitude}(F) &= \text{Sqrt}(\text{Power}(F))\end{aligned}$$

What does this really mean?

Before you panic, let me explain what this means in layman's terms. Given a time series, $x(n)$, you can determine if that time series contains a cosine component or a sine component at a given frequency, F , by doing the following:

- Create one new time series, $\cos(n)$, which is a cosine function with the frequency F .
- Create another new time series, $\sin(n)$, which is a sine function with the frequency F .
- Multiply $x(n)$ by $\cos(n)$ on a point by point basis and compute the sum of the products. Save this value, calling it $\text{Real}(F)$. This is an estimate of the amplitude, if any, of the cosine component with the matching frequency contained in the time series $x(n)$.
- Multiply $x(n)$ by $\sin(n)$ on a point by point basis and compute the sum of the products. Save this value, calling it $\text{Imag}(f)$. This is an estimate of the amplitude, if any, of the sine component with the matching frequency contained in the time series $x(n)$.
- Consider the values for $\text{Real}(F)$ and $\text{Imag}(F)$ to be the real and imaginary parts of a complex number.
- Consider the sum of the squares of the real and imaginary parts to represent the power at that frequency in the time series.
- Consider the square root of the power to be the amplitude at that frequency in the time series. (This is the value that is plotted in [Figure 9](#),

[Figure 11](#), and [Figure 14](#).)

Compute the complex energy at each frequency

That is all there is to it. For each frequency of interest, you can use this process to compute a complex number, $\text{Real}(F) - j\text{Imag}(F)$, which represents the complex energy corresponding to that frequency in the target time series.

Similarly, you can compute the sum of the squares of the real and imaginary parts and consider that to be a measure of the power at that frequency in the time series. The square root of the power is the amplitude of the energy at that frequency.

Nested for loops

Normally we are interested in more than one frequency, so we would repeat the above procedure once for each frequency of interest. This suggests the use of nested **for** loops in the algorithm. The outer loop specifies the frequency of interest. The inner loop computes the sum of the products at a particular frequency.

Description of the transform method

The static method named **transform** performs a real to complex Fourier transform. The method does not implement the FFT algorithm. Rather, it implements a straightforward sampled data version of the continuous Fourier transform defined using integral calculus. (See *ForwardRealToComplexFFT01* for an FFT algorithm.)

The return values

The method returns the following:

- Real part of the spectral analysis result

- Imaginary part of the spectral analysis result
- Magnitude of the spectral analysis result
- Phase angle of the spectral analysis result in degrees

The transform method parameters

The method parameters are:

- double[] data - incoming real data
- double[] realOut - outgoing real data
- double[] imagOut - outgoing imaginary data
- double[] angleOut - outgoing phase angle in degrees
- double[] magnitude - outgoing amplitude spectrum
- int zero - the index of the incoming data sample that represents zero time
- double lowF - low frequency limit for computation as a fraction of sampling frequency
- double highF - high frequency limit for computation as a fraction of sampling frequency

Frequency increment, magnitude spectrum, and number of returned values

The computational frequency increment is the difference between the high and low limits divided by the length of the magnitude array.

The magnitude or amplitude is computed as the square root of the sum of the squares of the real and imaginary parts. This value is divided by the incoming data length, which is given by **data.length** .

The method returns a number of points in the frequency domain equal to the incoming data length regardless of the high and low frequency limits.

The beginning of the transform method

The class and the **transform** method begin in [Listing 14](#). The code in [Listing 14](#) is described above.

Listing 14. The beginning of the transform method.

```
public class ForwardRealToComplex01{

    public static void transform(
                                double[] data,
                                double[] realOut,
                                double[] imagOut,
                                double[] angleOut,
                                double[]
magnitude,
                                int zero,
                                double lowF,
                                double highF){
        double pi = Math.PI;//for convenience
        int dataLen = data.length;
        double delf = (highF-lowF)/data.length;
```

The remainder of the method and the class

The nested **for** loops discussed above are included in the code shown in [Listing 15](#). As suggested above, the outer loop iterates on frequency while the inner loop iterates on the values that make up the incoming samples. The code in the inner loop computes the sum of the product of the time series and the reference cosine and sine functions.

Listing 15. The remainder of the method and the class.

```
//Outer loop iterates on frequency
```

Listing 15. The remainder of the method and the class.

```
// values.
for(int i=0; i < dataLen;i++){
    double freq = lowF + i*delF;
    double real = 0.0;
    double imag = 0.0;
    double ang = 0.0;
    //Inner loop iterates on time-
    // series points.
    for(int j=0; j < dataLen; j++){
        real += data[j]*Math.cos(
                                2*pi*freq*(j-
zero));
        imag += data[j]*Math.sin(
                                2*pi*freq*(j-
zero));
    }//end inner loop

    realOut[i] = real/dataLen;
    imagOut[i] = imag/dataLen;
    magnitude[i] = (Math.sqrt(
                    real*real +
imag*imag))/dataLen;

    //Calculate and return the phase
    // angle in degrees.
    if(imag == 0.0 && real == 0.0){ang = 0.0;}
    else{ang = Math.atan(imag/real)*180.0/pi;}

    if(real < 0.0 && imag == 0.0){ang =
180.0;}
    else if(real < 0.0 && imag == -0.0){
        ang =
-180.0;}
    else if(real < 0.0 && imag > 0.0){
        ang +=
180.0;}
```

Listing 15. The remainder of the method and the class.

```
        else if(real < 0.0 && imag < 0.0){
                                ang +=
-180.0;}
        angleOut[i] = ang;
    }//end outer loop
} //end transform method

} //end class ForwardRealToComplex01
```

Store results in output array objects

At the end of each iteration of the inner loop, code in the outer loop deposits the real, imaginary, magnitude, and phase angle results in the output array objects. To accomplish this, the code:

- Computes the magnitude or amplitude as the square root of the sum of the squares of the real and imaginary parts.
- Performs some trigonometry operations to determine the phase angle in degrees based on the values of the real and imaginary parts.

Now you know about the DFT algorithm

Now you know about the DFT algorithm. You also know about some of the fundamental aspects of spectral analysis involving the sampling frequency and the folding frequency.

Future modules will discuss other aspects of spectral analysis including:

- Frequency resolution versus data length.
- The relationship between the phase angle and delays in the time domain.
- The reversible nature of the Fourier transform involving both forward and inverse Fourier transforms.

Spectral analysis using an FFT algorithm

At this point, I will present a similar spectral analysis program that uses an FFT algorithm. I will present this program with very little discussion. I am providing it in this module for two primary purposes:

- To allow you to experiment and appreciate the flexibility of the DFT as compared to the FFT.
- To allow you to experiment and appreciate the speed of the FFT as compared to the DFT.

The program named Dsp030

The program named **Dsp030** is very similar to **Dsp028** . The major differences are:

- Because it uses an FFT algorithm, **Dsp030** is much less flexible than **Dsp028** , particularly with respect to data length and selection of the frequencies of interest.
- Because it uses an FFT algorithm, **Dsp030** is much faster than **Dsp028** , particularly when used to perform spectral analysis on long data lengths.

A complete listing of **Dsp030** is provided in [Listing 22](#).

Description of the program named Dsp030

This program uses an FFT algorithm to compute and display the magnitude of the spectral content for up to five sinusoids having different frequencies and amplitudes. *(See the program named **Dsp028** for a program that does not use an FFT algorithm.)*

The input parameters

The program gets input parameters from a file named **Dsp030.txt** . If that file doesn't exist in the current directory, the program uses a set of default

parameters.

Each parameter value must be stored as characters on a separate line in the file named **Dsp030.txt** . The required input parameters are shown in [Figure 16](#). *(Contrast this with the required input parameters for **Dsp028** shown in [Figure 12](#).)*

Figure 16. Required input parameters for Dsp030.

Data length as type int (must be a power of 2)
Number of spectra as type int. Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

Note that in contrast with [Figure 12](#), the required input parameters for **Dsp030** do not include the sample number representing zero time, the lower frequency bound for computation of the spectra, and the upper frequency bound for computation of the spectra.

(The computational frequency range cannot be specified for the FFT algorithm. It always computes the spectra from zero to one unit less than the sampling frequency.)

Restrictions on the data length

Note also that the data length must always be a power of two. Otherwise, the FFT algorithm will fail to run properly.

(This restriction is an important contributor to the speed achieved by the FFT algorithm.)

The sinusoidal frequency values

As with **Dsp028**, the number of values in each of the lists must match the value for the number of spectra.

All frequency values are specified as a **double** representing a fractional part of the sampling frequency.

[Figure 17](#) shows the parameters used to produce the spectral analysis plots shown later in [Figure 18](#).

(Note that the data length is a power of two as required by the FFT.)

Figure 17. Example input parameters.

Figure 17. Example input parameters.

```
256
5
0.1
0.2
0.3
0.5
0.005
90
90
90
90
90
```

The plotting program

The plotting program that is used to plot the output data from this program requires that the program implement **GraphIntfc01** . For example, the plotting program named **Graph03** can be used to plot the data produced by this program. This requires that you enter the following at the command line prompt after everything is compiled:

```
java Graph03 Dsp030
```

The plotting program named **Graph06** can also be used to plot the data produced by this program, requiring that you enter the following at the command line prompt:

```
java Graph06 Dsp030
```

The transform method

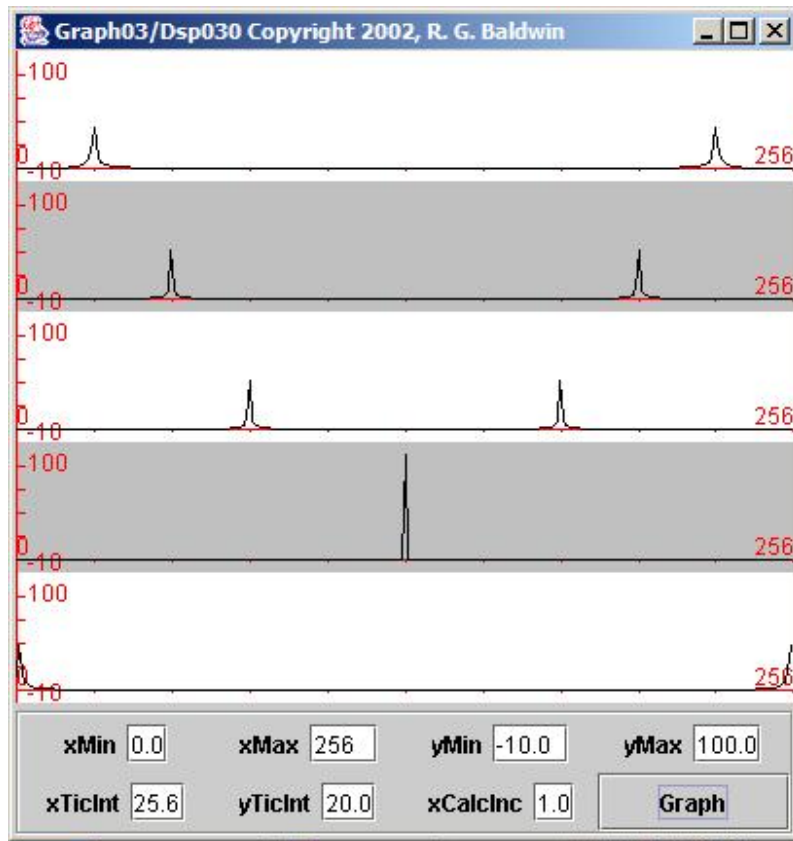
A static method named **transform** belonging to the class named **ForwardRealToComplexFFT01** is used to perform the actual spectral analysis. The method named **transform** implements an FFT algorithm. As mentioned above, the FFT algorithm requires that the data length be a power of two. This method will be discussed very briefly later.

A sample FFT spectral analysis

The output produced by running **Dsp030** using the input parameters shown in [Figure 17](#) is shown in [Figure 18](#).

Figure 18. FFT of five sinusoids.

Figure 18. FFT of five sinusoids.



Nothing special here

There is nothing special about this particular spectral analysis. I presented it here to illustrate the use of the FFT algorithm for spectral analysis. You should be able to produce the same results using the same program and the same parameters.

A matching DFT spectral analysis

[Figure 19](#) shows the parameters required for the program named **Dsp028** to perform a DFT spectral analysis producing the same results as those produced

by the FFT analysis shown in [Figure 18](#). Note that the data length has been set to 256 and the computational frequency range extends from zero to the sampling frequency in [Figure 19](#).

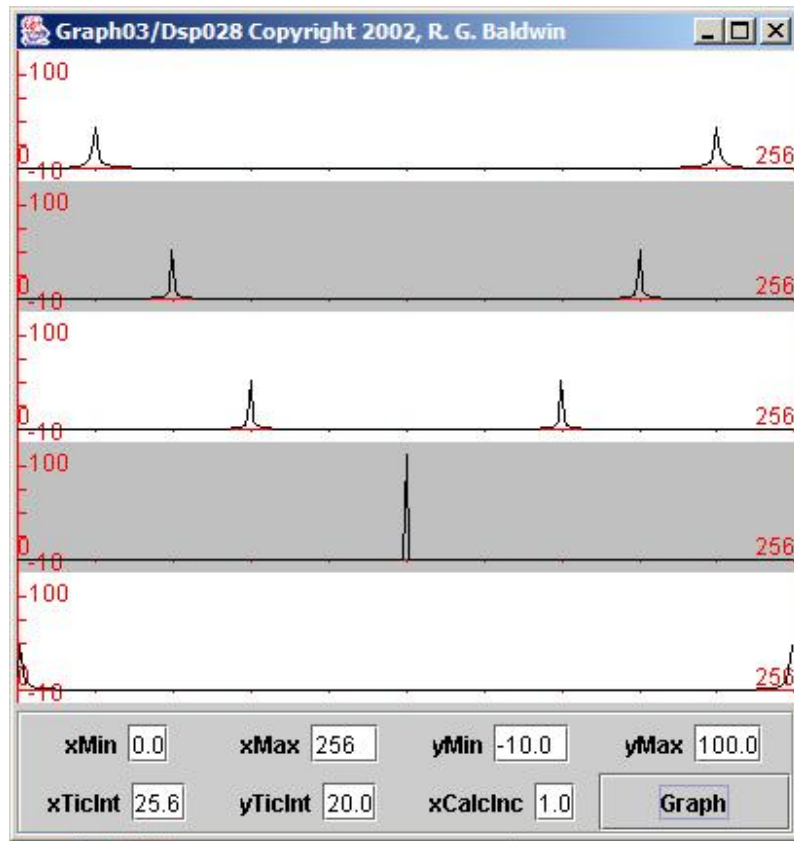
Figure 19. Parameters for A matching DFT spectral analysis.

```
Data length: 256
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 1.0
Number spectra: 5
Frequencies
0.1
0.2
0.3
0.5
0.0050
Amplitudes
90.0
90.0
90.0
90.0
90.0
```

The matching DFT output

The DFT output produced by running Dsp028 with the parameters shown in [Figure 19](#) is shown in [Figure 20](#).

Figure 20. DFT of five sinusoids.



Hopefully you noticed that [Figure 20](#) looks almost exactly like [Figure 18](#). This is how it should be. The DFT algorithm and the FFT algorithm are simply two different algorithms for computing the same results. However, the DFT algorithm is much more flexible than the FFT algorithm while the FFT algorithm is much faster than the DFT algorithm.

Repeat these two experiments

I recommend that you repeat these two experiments several times increasing the data length to a higher power of two each time you run the experiments.

On my machine, the DFT algorithm used by **Dsp028** becomes noticeably slow by the time the data length reaches 2048 samples. However, the FFT

algorithm used by **Dsp030** is still reasonably responsive at a data length of 131,072 samples.

(Performing the DFT on five input samples each having a data length of 131,072 samples would require an intolerably long time on my machine.)

If what you need is speed for long data lengths, the FFT is your best approach. On the other hand, if you need more flexibility than the FFT provides and the data length is not too long, then the DFT may be your best approach.

The ForwardRealToComplexFFT01 class

The **ForwardRealToComplexFFT01** class containing the method that implements the FFT algorithm is provided in [Listing 23](#) near the end of this module.

The FFT algorithm is based on some very complicated signal processing concepts. I'm not going to explain how this algorithm works in this module because I haven't given you the proper background for understanding it. I plan to explain additional signal processing concepts in future modules that will prepare you to understand how the FFT algorithm works.

Fortunately, you don't have to understand the mechanics of the FFT algorithm works to be able to use it.

Run the programs

I encourage you to copy, compile, and run the programs provided in this module. Experiment with them, making changes and observing the results of your changes.

I suggest that you begin by compiling and running the following files to confirm that everything is working correctly on your machine before

attempting to compile and run the spectral analysis programs:

- Dsp029.java
- GraphIntfc01.java
- Graph06.java

Make sure that you create an appropriate file named **Dsp029.txt** , as described in [Figure 2](#) . You should be able to reproduce my results if everything is working correctly.

Once you confirm that things are working correctly, copy, compile, and run the spectral analysis programs. Experiment with the parameters and try to understand the result of making changes to the parameters. Confirm the flexibility of the DFT algorithm and the speed of the FFT algorithm.

Summary

In this module I have provided and explained programs that illustrate the impact of sampling and the Nyquist folding frequency.

I have also provided and explained several different programs used for performing spectral analysis. The first program was a very general program that implements a *Discrete Fourier Transform (DFT)* algorithm. I explained this program in detail.

The second program was a less general, but much faster program that implements a *Fast Fourier Transform (FFT)* algorithm. I will defer an explanation of this program until a future module. I provided it in this module so that you can use it and compare it with the DFT program in terms of speed and flexibility.

What's next?

Future modules will discuss other aspects of spectral analysis including:

- Frequency resolution versus data length.
- The relationship between the phase angle and delays in the time domain.

- The reversible nature of the Fourier transform involving both forward and inverse Fourier transforms.
- Additional material aimed towards an understanding of the signal processing concepts behind the FFT algorithm.

Complete program listings

Complete listings of all the programs discussed in this module follow.

Listing 16. Dsp029.java.

```
/* File Dsp029.java  
Copyright 2004, R.G.Baldwin  
Rev 5/6/04
```

Generates and displays up to five sinusoids having different frequencies and amplitudes. Very useful for providing a visual illustration of the way in which frequencies above half the sampling frequency fold back down into the area bounded by zero and half the sampling frequency (the Nyquist folding frequency).

Gets input parameters from a file named Dsp029.txt. If that file doesn't exist in the current directory, the program uses a set of default parameters.

Each parameter value must be stored as characters on a separate line in the file named Dsp029.txt. The required parameters are as follows:

Listing 16. Dsp029.java.

Data length as type int
Number of sinusoids as type int. Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

The number of values in each of the lists must match the value for the number of spectra.

Note: All frequency values are specified as a double representing a fractional part of the sampling frequency.

Here is a set of sample parameter values. Don't allow blank lines at the end of the data in the file.

```
400.0
5
0.1
0.9
1.1
1.9
2.1
90
90
90
90
90
```

The plotting program that is used to plot the output data from this program requires that the program implement GraphIntfc01. For example, the plotting program named Graph06 can be used to plot the data produced by this program. When it is used, the usage information is:

Listing 16. Dsp029.java.

```
java Graph06 Dsp029
```

Tested using SDK 1.4.2 under WinXP.

```
*****/
```

```
import java.util.*;
```

```
import java.io.*;
```

```
class Dsp029 implements GraphIntf01{
```

```
    final double pi = Math.PI;//for simplification
```

```
    //Begin default parameters
```

```
    int len = 400;//data length
```

```
    int numberSinusoids = 5;
```

```
    //Frequencies of the sinusoids
```

```
    double[] freq = {0.1,0.25,0.5,0.75,0.9};
```

```
    //Amplitudes of the sinusoids
```

```
    double[] amp = {75,75,75,75,75};
```

```
    //End default parameters
```

```
    //Following arrays will be populated with
```

```
    // sinusoidal data to be plotted
```

```
    double[] data1 = new double[len];
```

```
    double[] data2 = new double[len];
```

```
    double[] data3 = new double[len];
```

```
    double[] data4 = new double[len];
```

```
    double[] data5 = new double[len];
```

```
    public Dsp029(){//constructor
```

```
        //Get the parameters from a file named
```

```
        // Dsp029.txt. Use the default parameters
```

```
        // if the file doesn't exist in the current
```

```
        // directory.
```

```
        if(new File("Dsp029.txt").exists()){
```

```
            getParameters();
```

```
        }//end if
```

Listing 16. Dsp029.java.

```
//Note that this program always generates
// five sinusoids, even if fewer than five
// were requested as the input parameter
// for numberSinusoids. In that case, the
// extras are generated using default values
// and simply ignored when the results are
// plotted.

//Create the raw data. Note that the
// argument for a sinusoid at half the
// sampling frequency would be (2*pi*x*0.5).
// This would represent one half cycle or pi
// radians per sample.
for(int n = 0;n < len;n++){
    data1[n] = amp[0]*Math.cos(2*pi*n*freq[0]);
    data2[n] = amp[1]*Math.cos(2*pi*n*freq[1]);
    data3[n] = amp[2]*Math.cos(2*pi*n*freq[2]);
    data4[n] = amp[3]*Math.cos(2*pi*n*freq[3]);
    data5[n] = amp[4]*Math.cos(2*pi*n*freq[4]);
} //end for loop

} //end constructor
//-----//

//This method gets processing parameters from
// a file named Dsp029.txt and stores those
// parameters in instance variables belonging
// to the object of type Dsp029.
void getParameters(){
    int cnt = 0;
    //Temporary holding area for strings. Allow
    // space for a few blank lines at the end
    // of the data in the file.
    String[] data = new String[20];
    try{
```

Listing 16. Dsp029.java.

```
//Open an input stream.
BufferedReader inData =
    new BufferedReader(new FileReader(
        "Dsp029.txt"));
//Read and save the strings from each of
// the lines in the file. Be careful to
// avoid having blank lines at the end,
// which may cause an ArrayIndexOutOfBoundsException
// exception to be thrown.
while((data[cnt] =
    inData.readLine()) != null){
    cnt++;
} //end while
inData.close();
} catch(IOException e){}

//Move the parameter values from the
// temporary holding array into the instance
// variables, converting from characters to
// numeric values in the process.
cnt = 0;
len = (int)Double.parseDouble(data[cnt++]);
numberSinusoids = (int)Double.parseDouble(
    data[cnt++]);
for(int fCnt = 0; fCnt < numberSinusoids;
    fCnt++){
    freq[fCnt] = Double.parseDouble(
        data[cnt++]);
} //end for loop

for(int aCnt = 0; aCnt < numberSinusoids;
    aCnt++){
    amp[aCnt] = Double.parseDouble(
        data[cnt++]);
} //end for loop
```

Listing 16. Dsp029.java.

```
//Print parameter values.
System.out.println();
System.out.println("Data length: " + len);
System.out.println(
    "Number sinusoids: " + numberSinusoids);
System.out.println("Frequencies");
for(cnt = 0;cnt < numberSinusoids;cnt++){
    System.out.println(freq[cnt]);
} //end for loop
System.out.println("Amplitudes");
for(cnt = 0;cnt < numberSinusoids;cnt++){
    System.out.println(amp[cnt]);
} //end for loop

} //end getParameters
//-----//
//The following six methods are required by the
// interface named GraphIntfc01. The plotting
// program pulls the data values to be plotted
// by calling these methods.
public int getNmbr(){
    //Return number of functions to
    // process. Must not exceed 5.
    return numberSinusoids;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > data1.length-1){
        return 0;
    }else{
        return data1[index];
    } //end else
} //end function
//-----//
```

Listing 16. Dsp029.java.

```
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > data2.length-1){
        return 0;
    }else{
        return data2[index];
    }//end else
}//end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > data3.length-1){
        return 0;
    }else{
        return data3[index];
    }//end else
}//end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > data4.length-1){
        return 0;
    }else{
        return data4[index];
    }//end else
}//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > data5.length-1){
        return 0;
    }else{
```

Listing 16. Dsp029.java.

```
        return data5[index];
    }//end else
} //end function
//-----//

} //end class Dsp029
```

Listing 17. GraphIntfc01.java.

```
/* File GraphIntfc01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04
```

This interface must be implemented by classes whose objects produce data to be plotted by programs such as Graph03 and Graph06.

Tested using SDK 1.4.2 under WinXP.

```
*****/
```

```
public interface GraphIntfc01{
    public int getNmbr();
    public double f1(double x);
    public double f2(double x);
    public double f3(double x);
    public double f4(double x);
    public double f5(double x);
} //end GraphIntfc01
```


Listing 18. Graph06.java.

```
/* File Graph06.java  
Copyright 2002, R.G.Baldwin  
Revised 5/15/04
```

Very similar to Graph03, except that each point is displayed as a rectangle, centered on the sample. Can be used to explain integration through summation of the sample values.

Note: This program requires access to the interface named GraphIntfc01.

This is a plotting program. It is designed to access a class file, which implements GraphIntfc01, and to plot up to five functions defined in that class file. The plotting surface is divided into the required number of equally sized plotting areas, and one function is plotted on cartesian coordinates in each area.

The methods corresponding to the functions are named f1, f2, f3, f4, and f5.

The class containing the functions must also define a method named getNmbr(), which takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a NoSuchMethodException will be thrown.

Listing 18. Graph06.java.

Note that the constructor for the class that implements GraphIntf01 must not require any parameters due to the use of the newInstance method of the Class class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with f5 and work down toward f1. For example, if the number of functions is 3, then the program will expect to call methods named f1, f2, and f3. It is OK for the absent methods to be defined in the class. They simply won't be called.

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

The cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample

Listing 18. Graph06.java.

class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named junk and plots the data provided by that class. Thus, you can test the program by running it with no command-line parameter.

This program provides the following text fields for user input, along with a button labeled Graph. This allows the user to adjust the parameters and replot the graph as many times with as many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these parameters and then click the Graph button to cause the five functions to be re-plotted according to the new parameters.

Listing 18. Graph06.java.

Whenever the Graph button is clicked, the event handler instantiates a new object of the class that implements the GraphIntfc01 interface. Depending on the nature of that class, this may be redundant in some cases. However, it is useful in those cases where it is necessary to refresh the values of instance variables defined in the class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were first defined in the Color class of v1.4.0. Therefore, the program requires v1.4.0 or later to compile and run correctly.

*****/

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph06{
    public static void main(
        String[] args)
        throws NoSuchElementException,
            ClassNotFoundException,
            InstantiationException,
            IllegalAccessException{
        if(args.length == 1){
            //pass command-line paramater
```

Listing 18. Graph06.java.

```
        new GUI(args[0]);
    }else{
        //no command-line parameter given
        new GUI(null);
    }//end else
} // end main
} //end class Graph06 definition
//=====//

class GUI extends JFrame
    implements ActionListener{

    //Define plotting parameters and
    // their default values.
    double xMin = 0.0;
    double xMax = 400.0;
    double yMin = -100.0;
    double yMax = 100.0;

    //Tic mark intervals
    double xTicInt = 20.0;
    double yTicInt = 20.0;

    //Tic mark lengths. If too small
    // on x-axis, a default value is
    // used later.
    double xTicLen = (yMax-yMin)/50;
    double yTicLen = (xMax-xMin)/50;

    //Calculation interval along x-axis
    double xCalcInc = 1.0;

    //Text fields for plotting parameters
    JTextField xMinTxt =
        new JTextField("" + xMin);
    JTextField xMaxTxt =
```

Listing 18. Graph06.java.

```
        new JTextField("" + xMax);
JTextField yMinTxt =
        new JTextField("" + yMin);
JTextField yMaxTxt =
        new JTextField("" + yMax);
JTextField xTicIntTxt =
        new JTextField("" + xTicInt);
JTextField yTicIntTxt =
        new JTextField("" + yTicInt);
JTextField xCalcIncTxt =
        new JTextField("" + xCalcInc);

//Panels to contain a label and a
// text field
JPanel pan0 = new JPanel();
JPanel pan1 = new JPanel();
JPanel pan2 = new JPanel();
JPanel pan3 = new JPanel();
JPanel pan4 = new JPanel();
JPanel pan5 = new JPanel();
JPanel pan6 = new JPanel();

//Misc instance variables
int frmWidth = 408;
int frmHeight = 430;
int width;
int height;
int number;
GraphIntfc01 data;
String args = null;

//Plots are drawn on the canvases
// in this array.
Canvas[] canvases;

//Constructor
```

Listing 18. Graph06.java.

```
GUI(String args)throws
        NoSuchMethodException,
        ClassNotFoundException,
        InstantiationException,
        IllegalAccessException{

    if(args != null){
        //Save for use later in the
        // ActionEvent handler
        this.args = args;
        //Instantiate an object of the
        // target class using the String
        // name of the class.
        data = (GraphIntfc01)
                Class.forName(args).
                    newInstance();
    }else{
        //Instantiate an object of the
        // test class named junk.
        data = new junk();
    }//end else

    //Create array to hold correct
    // number of Canvas objects.
    canvases =
        new Canvas[data.getNmbr()];

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
        throw new NoSuchMethodException(
            "Too many functions. "
            + "Only 5 allowed.");
    }//end if
```

Listing 18. Graph06.java.

```
//Create the control panel and
// give it a border for cosmetics.
JPanel ctlPnl = new JPanel();
ctlPnl.setLayout(//?rows x 4 cols
                 new GridLayout(0,4));
ctlPnl.setBorder(
                 new EtchedBorder());

//Button for replotting the graph
JButton graphBtn =
    new JButton("Graph");
graphBtn.addActionListener(this);

//Populate each panel with a label
// and a text field. Will place
// these panels in a grid on the
// control panel later.
pan0.add(new JLabel("xMin"));
pan0.add(xMinTxt);

pan1.add(new JLabel("xMax"));
pan1.add(xMaxTxt);

pan2.add(new JLabel("yMin"));
pan2.add(yMinTxt);

pan3.add(new JLabel("yMax"));
pan3.add(yMaxTxt);

pan4.add(new JLabel("xTicInt"));
pan4.add(xTicIntTxt);

pan5.add(new JLabel("yTicInt"));
pan5.add(yTicIntTxt);

pan6.add(new JLabel("xCalcInc"));
```


Listing 18. Graph06.java.

```
pan6.add(xCalcIncTxt);

//Add the populated panels and the
// button to the control panel with
// a grid layout.
ctlPnl.add(pan0);
ctlPnl.add(pan1);
ctlPnl.add(pan2);
ctlPnl.add(pan3);
ctlPnl.add(pan4);
ctlPnl.add(pan5);
ctlPnl.add(pan6);
ctlPnl.add(graphBtn);

//Create a panel to contain the
// Canvas objects. They will be
// displayed in a one-column grid.
JPanel canvasPanel = new JPanel();
canvasPanel.setLayout(//?rows,1 col
                      new GridLayout(0,1));

//Create a custom Canvas object for
// each function to be plotted and
// add them to the one-column grid.
// Make background colors alternate
// between white and gray.
for(int cnt = 0;
    cnt < number; cnt++){
    switch(cnt){
        case 0 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.WHITE);
            break;
        case 1 :
```

Listing 18. Graph06.java.

```
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.LIGHT_GRAY);
        break;
    case 2 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.WHITE);

        break;
    case 3 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].setBackground(
            Color.LIGHT_GRAY);

        break;
    case 4 :
        canvases[cnt] =
            new MyCanvas(cnt);
        canvases[cnt].
            setBackground(Color.WHITE);
} //end switch
//Add the object to the grid.
canvasPanel.add(canvases[cnt]);
} //end for loop

//Add the sub-assemblies to the
// frame. Set its location, size,
// and title, and make it visible.
getContentPane().
    add(ctlPnl, "South");
getContentPane().
    add(canvasPanel, "Center");

setBounds(0, 0, frmWidth, frmHeight);
```

Listing 18. Graph06.java.

```
if(args == null){
    setTitle("Graph06, " +
            "Copyright 2002, " +
            "Richard G. Baldwin");
}else{
    setTitle("Graph06/" + args +
            " Copyright 2002, " +
            "R. G. Baldwin");
}//end else

setVisible(true);

//Set to exit on X-button click
setDefaultCloseOperation(
    EXIT_ON_CLOSE);

//Guarantee a repaint on startup.
for(int cnt = 0;
    cnt < number; cnt++){
    canvases[cnt].repaint();
}//end for loop

}//end constructor
//-----//

//This event handler is registered
// on the JButton to cause the
// functions to be replotted.
public void actionPerformed(
   (ActionEvent evt){
    //Re-instantiate the object that
    // provides the data
    try{
        if(args != null){
            data = (GraphIntfc01)Class.
```

Listing 18. Graph06.java.

```
        forName(args).newInstance();
    }else{
        data = new junk();
    }//end else
}catch(Exception e){
    //Known to be safe at this point.
    // Otherwise would have aborted
    // earlier.
};//end catch

//Set plotting parameters using
// data from the text fields.
xMin = Double.parseDouble(
        xMinTxt.getText());
xMax = Double.parseDouble(
        xMaxTxt.getText());
yMin = Double.parseDouble(
        yMinTxt.getText());
yMax = Double.parseDouble(
        yMaxTxt.getText());
xTicInt = Double.parseDouble(
        xTicIntTxt.getText());
yTicInt = Double.parseDouble(
        yTicIntTxt.getText());
xCalcInc = Double.parseDouble(
        xCalcIncTxt.getText());

//Calculate new values for the
// length of the tic marks on the
// axes. If too small on x-axis,
// a default value is used later.
xTicLen = (yMax-yMin)/50;
yTicLen = (xMax-xMin)/50;

//Repaint the plotting areas
for(int cnt = 0;
```

Listing 18. Graph06.java.

```
                cnt < number; cnt++){
    canvases[cnt].repaint();
} //end for loop

} //end actionPerformed
//-----//

//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
    int cnt; //object number
    //Factors to convert from double
    // values to integer pixel locations.
    double xScale;
    double yScale;

    MyCanvas(int cnt){ //save obj number
        this.cnt = cnt;
    } //end constructor

    //Override the paint method
    public void paint(Graphics g){
        //Get and save the size of the
        // plotting surface
        width = canvases[0].getWidth();
        height = canvases[0].getHeight();

        //Calculate the scale factors
        xScale = width/(xMax-xMin);
        yScale = height/(yMax-yMin);

        //Set the origin based on the
        // minimum values in x and y
        g.translate((int)((0-xMin)*xScale),
```

Listing 18. Graph06.java.

```
                (int)((0-yMin)*yScale));
drawAxes(g); //Draw the axes
g.setColor(Color.BLACK);

//Get initial data values
double xVal = xMin;
int oldX = getTheX(xVal);
int oldY = 0;
//Use the Canvas obj number to
// determine which method to
// call to get the value for y.
switch(cnt){
    case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
    case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
    case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
    case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
    case 4 :
        oldY = getTheY(data.f5(xVal));
} //end switch

//Now loop and plot the points
while(xVal < xMax){
    int yVal = 0;
    //Get next data value. Use the
    // Canvas obj number to
    // determine which method to
    // call to get the value for y.
    switch(cnt){
```

Listing 18. Graph06.java.

```
        case 0 :
            yVal =
                getTheY(data.f1(xVal));
            break;
        case 1 :
            yVal =
                getTheY(data.f2(xVal));
            break;
        case 2 :
            yVal =
                getTheY(data.f3(xVal));
            break;
        case 3 :
            yVal =
                getTheY(data.f4(xVal));
            break;
        case 4 :
            yVal =
                getTheY(data.f5(xVal));
    } //end switch1

    //Convert the x-value to an int
    // and draw the next horizontal
    // line segment

    int x = getTheX(xVal+xCalcInc/2);
    g.drawLine(oldX,yVal,x,yVal);

    //Draw a vertical line at the
    // old x-value
    int yZero = getTheY(0);
    g.drawLine(oldX,yZero,oldX,yVal);

    //Draw a vertical line at the
    // new y-value
```

Listing 18. Graph06.java.

```
        g.drawLine(x,yZero,x,yVal);

        //Increment along the x-axis
        xVal += xCalcInc;

        //Save end point to use as start
        // point for next line segment.
        oldX = x;
        oldY = yVal;
    } //end while loop

} //end overridden paint method
//-----//

//Method to draw axes with tic marks
// and labels in the color RED
void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Label left x-axis and bottom
    // y-axis. These are the easy
    // ones. Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                  getTheX(xMin),
                  getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                  getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis. These are the hard
    // ones because the position must
    // be adjusted by the font size and
    // the number of characters.
```


Listing 18. Graph06.java.

[illegible]

Listing 18. Graph06.java.

```
(int)(r2d.getHeight());

//Label the positive x-axis and the
// positive y-axis using the width
// and height from above to
// position the labels. These
// labels apply to the very ends of
// the axes at the edge of the
// plotting surface.
g.drawString("" + (int)xMax,
             getTheX(xMax)-labWidth,
             getTheY(xTicLen/2)-2);
g.drawString("" + (int)yMax,
             getTheX(yTicLen/2)+2,
             getTheY(yMax)+labHeight);

//Draw the axes
g.drawLine(getTheX(xMin),
           getTheY(0.0),
           getTheX(xMax),
           getTheY(0.0));

g.drawLine(getTheX(0.0),
           getTheY(yMin),
           getTheX(0.0),
           getTheY(yMax));

//Draw the tic marks on axes
xTics(g);
yTics(g);
} //end drawAxes

//-----//

//Method to draw tic marks on x-axis
void xTics(Graphics g){
```

Listing 18. Graph06.java.

```
double xDoub = 0;
int x = 0;

//Get the ends of the tic marks.
int topEnd = getTheY(xTicLen/2);
int bottomEnd =
    getTheY(-xTicLen/2);

//If the vertical size of the
// plotting area is small, the
// calculated tic size may be too
// small. In that case, set it to
// 10 pixels.
if(topEnd < 5){
    topEnd = 5;
    bottomEnd = -5;
} //end if

//Loop and draw a series of short
// lines to serve as tic marks.
// Begin with the positive x-axis
// moving to the right from zero.
while(xDoub < xMax){
    x = getTheX(xDoub);
    g.drawLine(x, topEnd, x, bottomEnd);
    xDoub += xTicInt;
} //end while

//Now do the negative x-axis moving
// to the left from zero
xDoub = 0;
while(xDoub > xMin){
    x = getTheX(xDoub);
    g.drawLine(x, topEnd, x, bottomEnd);
    xDoub -= xTicInt;
} //end while
```

Listing 18. Graph06.java.

```
}//end xTics
//-----//

//Method to draw tic marks on y-axis
void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
        y = getTheY(yDoub);
        g.drawLine(rightEnd,y,leftEnd,y);
        yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
        y = getTheY(yDoub);
        g.drawLine(rightEnd,y,leftEnd,y);
        yDoub -= yTicInt;
    }//end while

}//end yTics

//-----//

//This method translates and scales
// a double y value to plot properly
```

Listing 18. Graph06.java.

```
// in the integer coordinate system.
// In addition to scaling, it causes
// the positive direction of the
// y-axis to be from bottom to top.
int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
} //end getTheY
//-----//

//This method scales a double x value
// to plot properly in the integer
// coordinate system.
int getTheX(double x){
    return (int)(x*xScale);
} //end getTheX
//-----//

} //end inner class MyCanvas
//=====//

} //end class GUI
//=====//

//Sample test class. Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
    public int getNmbr(){
        //Return number of functions to
        // process. Must not exceed 5.
        return 4;
    } //end getNmbr

    public double f1(double x){
```

Listing 18. Graph06.java.

```
        return (x*x*x)/200.0;
    }//end f1

    public double f2(double x){
        return -(x*x*x)/200.0;
    }//end f2

    public double f3(double x){
        return (x*x)/200.0;
    }//end f3

    public double f4(double x){
        return 50*Math.cos(x/10.0);
    }//end f4

    public double f5(double x){
        return 100*Math.sin(x/20.0);
    }//end f5

} //end sample class junk
```

Listing 19. Dsp028.java.

```
/* File Dsp028.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04
```

Computes and displays the magnitude of the spectral content for up to five sinusoids having

Listing 19. Dsp028.java.

different frequencies and amplitudes.

Gets input parameters from a file named Dsp028.txt. If that file doesn't exist in the current directory, the program uses a set of default parameters.

Each parameter value must be stored as characters on a separate line in the file named Dsp028.txt. The required parameters are as follows:

Data length as type int
Sample number representing zero time as type int
Lower frequency bound as type double (See note)
Upper frequency bound as type double
Number of spectra as type int. Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

The number of values in each of the lists must match the value for the number of spectra.

Note: All frequency values are specified as a double representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is half the sampling frequency.

Here is a set of sample parameter values. Don't allow blank lines at the end of the data in the file.

```
400
0
0.0
1.0
```

Listing 19. Dsp028.java.

```
5
0.1
0.2
0.3
0.4
0.45
60
70
80
90
100
```

The plotting program that is used to plot the output data from this program requires that the program implement `GraphIntfc01`. For example, the plotting program named `Graph03` can be used to plot the data produced by this program. When it is used, the usage information is:

```
java Graph03 Dsp028
```

A static method named `transform` belonging to the class named `ForwardRealToComplex01` is used to perform the actual spectral analysis. The method named `transform` does not implement an FFT algorithm. Rather, it is more general than, but much slower than an FFT algorithm. (See the program named `Dsp030` for the use of an FFT algorithm.)

Tested using SDK 1.4.2 under WinXP.

```
*****/
import java.util.*;
import java.io.*;

class Dsp028 implements GraphIntfc01{
```


Listing 19. Dsp028.java.

```
final double pi = Math.PI;//for simplification

//Begin default parameters
int len = 400;//data length
//Sample that represents zero time.
int zeroTime = 0;
//Low and high frequency limits for the
// spectral analysis.
double lowF = 0.0;
double highF = 1.0;
int numberSpectra = 5;
//Frequencies of the sinusoids
double[] freq = {0.1,0.2,0.3,0.4,0.5};
//Amplitudes of the sinusoids
double[] amp = {60,70,80,90,100};
//End default parameters

//Following arrays will contain data that is
// input to the spectral analysis process.
double[] data1;
double[] data2;
double[] data3;
double[] data4;
double[] data5;

//Following arrays receive information back
// from the spectral analysis that is not used
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] magnitude1;
```

Listing 19. Dsp028.java.

```
double[] magnitude2;
double[] magnitude3;
double[] magnitude4;
double[] magnitude5;

public Dsp028(){//constructor

    //Get the parameters from a file named
    // Dsp028.txt. Use the default parameters
    // if the file doesn't exist in the current
    // directory.
    if(new File("Dsp028.txt").exists()){
        getParameters();
    }//end if

    //Note that this program always processes
    // five sinusoids, even if fewer than five
    // were requested as the input parameter
    // for numberSpectra. In that case, the
    // extras are processed using default values
    // and simply ignored when the results are
    // plotted.

    //Create the raw data. Note that the
    // argument for a sinusoid at half the
    // sampling frequency would be (2*pi*x*0.5).
    // This would represent one half cycle or pi
    // radians per sample.
    //First create empty array objects.
    double[] data1 = new double[len];
    double[] data2 = new double[len];
    double[] data3 = new double[len];
    double[] data4 = new double[len];
    double[] data5 = new double[len];
    //Now populate the array objects
    for(int n = 0;n < len;n++){
```

Listing 19. Dsp028.java.

```
        data1[n] = amp[0]*Math.cos(2*pi*n*freq[0]);
        data2[n] = amp[1]*Math.cos(2*pi*n*freq[1]);
        data3[n] = amp[2]*Math.cos(2*pi*n*freq[2]);
        data4[n] = amp[3]*Math.cos(2*pi*n*freq[3]);
        data5[n] = amp[4]*Math.cos(2*pi*n*freq[4]);
    } //end for loop
```

```
    //Compute magnitude spectra of the raw data
    // and save it in output arrays. Note that
    // the real, imag, and angle arrays are not
    // used later, so they are discarded each
    // time a new spectral analysis is performed.
```

```
    magnitude1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1, real,
        imag, angle, magnitude1, zeroTime, lowF, highF);
```

```
    magnitude2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2, real,
        imag, angle, magnitude2, zeroTime, lowF, highF);
```

```
    magnitude3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3, real,
        imag, angle, magnitude3, zeroTime, lowF, highF);
```

```
    magnitude4 = new double[len];
    real = new double[len];
    imag = new double[len];
```

Listing 19. Dsp028.java.

```
    angle = new double[len];
    ForwardRealToComplex01.transform(data4, real,
        imag, angle, magnitude4, zeroTime, lowF, highF);

    magnitude5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5, real,
        imag, angle, magnitude5, zeroTime, lowF, highF);
} //end constructor
//-----//

//This method gets processing parameters from
// a file named Dsp028.txt and stores those
// parameters in instance variables belonging
// to the object of type Dsp028.
void getParameters(){
    int cnt = 0;
    //Temporary holding area for strings. Allow
    // space for a few blank lines at the end
    // of the data in the file.
    String[] data = new String[20];
    try{
        //Open an input stream.
        BufferedReader inData =
            new BufferedReader(new FileReader(
                "Dsp028.txt"));
        //Read and save the strings from each of
        // the lines in the file. Be careful to
        // avoid having blank lines at the end,
        // which may cause an ArrayIndexOutOfBoundsException
        // exception to be thrown.
        while((data[cnt] =
            inData.readLine()) != null){
            cnt++;
        }
    }
```

Listing 19. Dsp028.java.

```
        }//end while
        inData.close();
    }catch(IOException e){}

    //Move the parameter values from the
    // temporary holding array into the instance
    // variables, converting from characters to
    // numeric values in the process.
    cnt = 0;
    len = (int)Double.parseDouble(data[cnt++]);
    zeroTime = (int)Double.parseDouble(
                                                data[cnt++]);
    lowF = Double.parseDouble(data[cnt++]);
    highF = Double.parseDouble(data[cnt++]);
    numberSpectra = (int)Double.parseDouble(
                                                data[cnt++]);
    for(int fCnt = 0;fCnt < numberSpectra;
                                                fCnt++){
        freq[fCnt] = Double.parseDouble(
                                                data[cnt++]);
    }//end for loop
    for(int aCnt = 0;aCnt < numberSpectra;
                                                aCnt++){
        amp[aCnt] = Double.parseDouble(
                                                data[cnt++]);
    }//end for loop

    //Print parameter values.
    System.out.println();
    System.out.println("Data length: " + len);
    System.out.println(
        "Sample for zero time: " + zeroTime);
    System.out.println(
        "Lower frequency bound: " + lowF);
    System.out.println(
        "Upper frequency bound: " + highF);
```

Listing 19. Dsp028.java.

```
        System.out.println(
            "Number spectra: " + numberSpectra);
    System.out.println("Frequencies");
    for(cnt = 0;cnt < numberSpectra;cnt++){
        System.out.println(freq[cnt]);
    }//end for loop
    System.out.println("Amplitudes");
    for(cnt = 0;cnt < numberSpectra;cnt++){
        System.out.println(amp[cnt]);
    }//end for loop

} //end getParameters
//-----//
//The following six methods are required by the
// interface named GraphIntfc01. The plotting
// program pulls the data values to be plotted
// by calling these methods.
public int getNmbr(){
    //Return number of functions to
    // process. Must not exceed 5.
    return numberSpectra;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude1.length-1){
        return 0;
    }else{
        return magnitude1[index];
    } //end else
} //end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
```

Listing 19. Dsp028.java.

```
        index > magnitude2.length-1){
            return 0;
        }else{
            return magnitude2[index];
        }//end else
    }//end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude3.length-1){
        return 0;
    }else{
        return magnitude3[index];
    }//end else
}//end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude4.length-1){
        return 0;
    }else{
        return magnitude4[index];
    }//end else
}//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude5.length-1){
        return 0;
    }else{
        return magnitude5[index];
    }//end else
}//end function
```

Listing 19. Dsp028.java.

```
//-----//  
  
} //end class Dsp028
```

Listing 20. Graph03.java.

```
/* File Graph03.java  
Copyright 2002, R.G.Baldwin
```

This program is very similar to Graph01 except that it has been modified to allow the user to manually resize and replot the frame.

Note: This program requires access to the interface named GraphIntfc01.

This is a plotting program. It is designed to access a class file, which implements GraphIntfc01, and to plot up to five functions defined in that class file. The plotting surface is divided into the required number of equally sized plotting areas, and one function is plotted on cartesian coordinates in each area.

The methods corresponding to the functions are named f1, f2, f3, f4,

Listing 20. Graph03.java.

and f5.

The class containing the functions must also define a method named `getNmbr()`, which takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a `NoSuchMethodException` will be thrown.

Note that the constructor for the class that implements `GraphIntfc01` must not require any parameters due to the use of the `newInstance` method of the `Class` class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with `f5` and work down toward `f1`. For example, if the number of functions is 3, then the program will expect to call methods named `f1`, `f2`, and `f3`. It is OK for the absent methods to be defined in the class. They simply won't be called.

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

Listing 20. Graph03.java.

The cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named junk and plots the data provided by that class. Thus, you can test the program by running it with no command-line parameter.

This program provides the following text fields for user input, along with a button labeled Graph. This allows the user to adjust the parameters and replot the graph as many times with as many plotting scales as needed:

```
xMin = minimum x-axis value  
xMax = maximum x-axis value  
yMin = minimum y-axis value
```

Listing 20. Graph03.java.

yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these parameters and then click the Graph button to cause the five functions to be re-plotted according to the new parameters.

Whenever the Graph button is clicked, the event handler instantiates a new object of the class that implements the GraphIntfc01 interface. Depending on the nature of that class, this may be redundant in some cases. However, it is useful in those cases where it is necessary to refresh the values of instance variables defined in the class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were first defined in the Color class of v1.4.0. Therefore, the program requires v1.4.0 or later to compile and run correctly.

*****/

```
import java.awt.*;  
import java.awt.event.*;  
import java.awt.geom.*;  
import javax.swing.*;  
import javax.swing.border.*;
```

Listing 20. Graph03.java.

```
class Graph03{
    public static void main(
        String[] args)
        throws NoSuchMethodException,
               ClassNotFoundException,
               InstantiationException,
               IllegalAccessException{
        if(args.length == 1){
            //pass command-line parameter
            new GUI(args[0]);
        }else{
            //no command-line parameter given
            new GUI(null);
        }//end else
    }// end main
}//end class Graph03 definition
//=====//

class GUI extends JFrame
    implements ActionListener{

    //Define plotting parameters and
    // their default values.
    double xMin = 0.0;
    double xMax = 400.0;
    double yMin = -100.0;
    double yMax = 100.0;

    //Tic mark intervals
    double xTicInt = 20.0;
    double yTicInt = 20.0;

    //Tic mark lengths. If too small
    // on x-axis, a default value is
    // used later.
```

Listing 20. Graph03.java.

```
double xTicLen = (yMax-yMin)/50;
double yTicLen = (xMax-xMin)/50;

//Calculation interval along x-axis
double xCalcInc = 1.0;

//Text fields for plotting parameters
JTextField xMinTxt =
    new JTextField("" + xMin);
JTextField xMaxTxt =
    new JTextField("" + xMax);
JTextField yMinTxt =
    new JTextField("" + yMin);
JTextField yMaxTxt =
    new JTextField("" + yMax);
JTextField xTicIntTxt =
    new JTextField("" + xTicInt);
JTextField yTicIntTxt =
    new JTextField("" + yTicInt);
JTextField xCalcIncTxt =
    new JTextField("" + xCalcInc);

//Panels to contain a label and a
// text field
JPanel pan0 = new JPanel();
JPanel pan1 = new JPanel();
JPanel pan2 = new JPanel();
JPanel pan3 = new JPanel();
JPanel pan4 = new JPanel();
JPanel pan5 = new JPanel();
JPanel pan6 = new JPanel();

//Misc instance variables
int frmWidth = 408;
int frmHeight = 430;
int width;
```

Listing 20. Graph03.java.

```
int height;
int number;
GraphIntfc01 data;
String args = null;

//Plots are drawn on the canvases
// in this array.
Canvas[] canvases;

//Constructor
GUI(String args)throws
    NoSuchMethodException,
    ClassNotFoundException,
    InstantiationException,
    IllegalAccessException{

    if(args != null){
        //Save for use later in the
        // ActionEvent handler
        this.args = args;
        //Instantiate an object of the
        // target class using the String
        // name of the class.
        data = (GraphIntfc01)
            Class.forName(args).
                newInstance();
    }else{
        //Instantiate an object of the
        // test class named junk.
        data = new junk();
    }//end else

    //Create array to hold correct
    // number of Canvas objects.
    canvases =
        new Canvas[data.getNمبر()];
```

Listing 20. Graph03.java.

```
//Throw exception if number of
// functions is greater than 5.
number = data.getNmbr();
if(number > 5){
    throw new NoSuchMethodException(
        "Too many functions. "
        + "Only 5 allowed.");
} //end if

//Create the control panel and
// give it a border for cosmetics.
JPanel ctlPnl = new JPanel();
ctlPnl.setLayout(//?rows x 4 cols
    new GridLayout(0,4));
ctlPnl.setBorder(
    new EtchedBorder());

//Button for replotting the graph
JButton graphBtn =
    new JButton("Graph");
graphBtn.addActionListener(this);

//Populate each panel with a label
// and a text field. Will place
// these panels in a grid on the
// control panel later.
pan0.add(new JLabel("xMin"));
pan0.add(xMinTxt);

pan1.add(new JLabel("xMax"));
pan1.add(xMaxTxt);

pan2.add(new JLabel("yMin"));
pan2.add(yMinTxt);
```

Listing 20. Graph03.java.

```
pan3.add(new JLabel("yMax"));
pan3.add(yMaxTxt);

pan4.add(new JLabel("xTicInt"));
pan4.add(xTicIntTxt);

pan5.add(new JLabel("yTicInt"));
pan5.add(yTicIntTxt);

pan6.add(new JLabel("xCalcInc"));
pan6.add(xCalcIncTxt);

//Add the populated panels and the
// button to the control panel with
// a grid layout.
ctlPnl.add(pan0);
ctlPnl.add(pan1);
ctlPnl.add(pan2);
ctlPnl.add(pan3);
ctlPnl.add(pan4);
ctlPnl.add(pan5);
ctlPnl.add(pan6);
ctlPnl.add(graphBtn);

//Create a panel to contain the
// Canvas objects. They will be
// displayed in a one-column grid.
JPanel canvasPanel = new JPanel();
canvasPanel.setLayout(//?rows,1 col
                      new GridLayout(0,1));

//Create a custom Canvas object for
// each function to be plotted and
// add them to the one-column grid.
// Make background colors alternate
// between white and gray.
```


Listing 20. Graph03.java.

```
for(int cnt = 0;
    cnt < number; cnt++){
    switch(cnt){
        case 0 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.WHITE);

            break;
        case 1 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.LIGHT_GRAY);

            break;
        case 2 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.WHITE);

            break;
        case 3 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].setBackground(
                Color.LIGHT_GRAY);

            break;
        case 4 :
            canvases[cnt] =
                new MyCanvas(cnt);
            canvases[cnt].
                setBackground(Color.WHITE);
    }//end switch
    //Add the object to the grid.
    canvasPanel.add(canvases[cnt]);
};//end for loop
```

Listing 20. Graph03.java.

```
//Add the sub-assemblies to the
// frame. Set its location, size,
// and title, and make it visible.
getContentPane().
    add(ctlPnl,"South");
getContentPane().
    add(canvasPanel,"Center");

setBounds(0,0,frmWidth,frmHeight);

if(args == null){
    setTitle("Graph03, " +
        "Copyright 2002, " +
        "Richard G. Baldwin");
}else{
    setTitle("Graph03/" + args +
        " Copyright 2002, " +
        "R. G. Baldwin");
}//end else

setVisible(true);

//Set to exit on X-button click
setDefaultCloseOperation(
    EXIT_ON_CLOSE);

//Guarantee a repaint on startup.
for(int cnt = 0;
    cnt < number; cnt++){
    canvases[cnt].repaint();
}//end for loop

}//end constructor
//-----//
```

Listing 20. Graph03.java.

```
//This event handler is registered
// on the JButton to cause the
// functions to be replotted.
public void actionPerformed(
                       (ActionEvent evt){
    //Re-instantiate the object that
    // provides the data
    try{
        if(args != null){
            data = (GraphIntfc01)Class.
                forName(args).newInstance();
        }else{
            data = new junk();
        }//end else
    }catch(Exception e){
        //Known to be safe at this point.
        // Otherwise would have aborted
        // earlier.
    }//end catch

    //Set plotting parameters using
    // data from the text fields.
    xMin = Double.parseDouble(
        xMinTxt.getText());
    xMax = Double.parseDouble(
        xMaxTxt.getText());
    yMin = Double.parseDouble(
        yMinTxt.getText());
    yMax = Double.parseDouble(
        yMaxTxt.getText());
    xTicInt = Double.parseDouble(
        xTicIntTxt.getText());
    yTicInt = Double.parseDouble(
        yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(
        xCalcIncTxt.getText());
```

Listing 20. Graph03.java.

```
//Calculate new values for the
// length of the tic marks on the
// axes.  If too small on x-axis,
// a default value is used later.
xTicLen = (yMax-yMin)/50;
yTicLen = (xMax-xMin)/50;

//Repaint the plotting areas
for(int cnt = 0;
        cnt < number; cnt++){
    canvases[cnt].repaint();
} //end for loop

} //end actionPerformed
//-----//

//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
    int cnt;//object number
    //Factors to convert from double
    // values to integer pixel locations.
    double xScale;
    double yScale;

    MyCanvas(int cnt){//save obj number
        this.cnt = cnt;
    } //end constructor

    //Override the paint method
    public void paint(Graphics g){

        //Get and save the size of the
```

Listing 20. Graph03.java.

```
// plotting surface
width = canvases[0].getWidth();
height = canvases[0].getHeight();

//Calculate the scale factors
xScale = width/(xMax-xMin);
yScale = height/(yMax-yMin);

//Set the origin based on the
// minimum values in x and y
g.translate((int)((0-xMin)*xScale),
            (int)((0-yMin)*yScale));
drawAxes(g); //Draw the axes
g.setColor(Color.BLACK);

//Get initial data values
double xVal = xMin;
int oldX = getTheX(xVal);
int oldY = 0;
//Use the Canvas obj number to
// determine which method to
// call to get the value for y.
switch(cnt){
    case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
    case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
    case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
    case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
    case 4 :
```

Listing 20. Graph03.java.

```
        oldY = getTheY(data.f5(xVal));
    }//end switch

    //Now loop and plot the points
    while(xVal < xMax){
        int yVal = 0;
        //Get next data value. Use the
        // Canvas obj number to
        // determine which method to
        // call to get the value for y.
        switch(cnt){
            case 0 :
                yVal =
                    getTheY(data.f1(xVal));
                break;
            case 1 :
                yVal =
                    getTheY(data.f2(xVal));
                break;
            case 2 :
                yVal =
                    getTheY(data.f3(xVal));
                break;
            case 3 :
                yVal =
                    getTheY(data.f4(xVal));
                break;
            case 4 :
                yVal =
                    getTheY(data.f5(xVal));
        }//end switch1

        //Convert the x-value to an int
        // and draw the next line segment
        int x = getTheX(xVal);
        g.drawLine(oldX,oldY,x,yVal);
```

Listing 20. Graph03.java.

```
//Increment along the x-axis
xVal += xCalcInc;

//Save end point to use as start
// point for next line segment.
oldX = x;
oldY = yVal;
} //end while loop

} //end overridden paint method
//-----//

//Method to draw axes with tic marks
// and labels in the color RED
void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Label left x-axis and bottom
    // y-axis. These are the easy
    // ones. Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                  getTheX(xMin),
                  getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                  getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis. These are the hard
    // ones because the position must
    // be adjusted by the font size and
    // the number of characters.
    //Get the width of the string for
```


Listing 20. Graph03.java.

```
//Label the positive x-axis and the
// positive y-axis using the width
// and height from above to
// position the labels. These
// labels apply to the very ends of
// the axes at the edge of the
// plotting surface.
g.drawString("" + (int)xMax,
             getTheX(xMax)-labWidth,
             getTheY(xTicLen/2)-2);
g.drawString("" + (int)yMax,
             getTheX(yTicLen/2)+2,
             getTheY(yMax)+labHeight);

//Draw the axes
g.drawLine(getTheX(xMin),
           getTheY(0.0),
           getTheX(xMax),
           getTheY(0.0));

g.drawLine(getTheX(0.0),
           getTheY(yMin),
           getTheX(0.0),
           getTheY(yMax));

//Draw the tic marks on axes
xTics(g);
yTics(g);
} //end drawAxes

//-----//

//Method to draw tic marks on x-axis
void xTics(Graphics g){
    double xDoub = 0;
```

Listing 20. Graph03.java.

```
int x = 0;

//Get the ends of the tic marks.
int topEnd = getTheY(xTicLen/2);
int bottomEnd =
    getTheY(-xTicLen/2);

//If the vertical size of the
// plotting area is small, the
// calculated tic size may be too
// small. In that case, set it to
// 10 pixels.
if(topEnd < 5){
    topEnd = 5;
    bottomEnd = -5;
}//end if

//Loop and draw a series of short
// lines to serve as tic marks.
// Begin with the positive x-axis
// moving to the right from zero.
while(xDoub < xMax){
    x = getTheX(xDoub);
    g.drawLine(x,topEnd,x,bottomEnd);
    xDoub += xTicInt;
}//end while

//Now do the negative x-axis moving
// to the left from zero
xDoub = 0;
while(xDoub > xMin){
    x = getTheX(xDoub);
    g.drawLine(x,topEnd,x,bottomEnd);
    xDoub -= xTicInt;
}//end while
```

Listing 20. Graph03.java.

```
//end xTics
//-----//

//Method to draw tic marks on y-axis
void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
        y = getTheY(yDoub);
        g.drawLine(rightEnd,y,leftEnd,y);
        yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
        y = getTheY(yDoub);
        g.drawLine(rightEnd,y,leftEnd,y);
        yDoub -= yTicInt;
    }//end while

} //end yTics

//-----//

//This method translates and scales
// a double y value to plot properly
// in the integer coordinate system.
```

Listing 20. Graph03.java.

```
// In addition to scaling, it causes
// the positive direction of the
// y-axis to be from bottom to top.
int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
} //end getTheY
//-----//

//This method scales a double x value
// to plot properly in the integer
// coordinate system.
int getTheX(double x){
    return (int)(x*xScale);
} //end getTheX
//-----//

} //end inner class MyCanvas
//=====//

} //end class GUI
//=====//

//Sample test class. Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
    public int getNmbr(){
        //Return number of functions to
        // process. Must not exceed 5.
        return 4;
    } //end getNmbr

    public double f1(double x){
        return (x*x*x)/200.0;
    }
}
```

Listing 20. Graph03.java.

```
//end f1

public double f2(double x){
    return -(x*x*x)/200.0;
} //end f2

public double f3(double x){
    return (x*x)/200.0;
} //end f3

public double f4(double x){
    return 50*Math.cos(x/10.0);
} //end f4

public double f5(double x){
    return 100*Math.sin(x/20.0);
} //end f5

} //end sample class junk
```

Listing 21. ForwardRealToComplex01.java.

```
/*File ForwardRealToComplex01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04
```

The static method named transform performs a real to complex Fourier transform.

Listing 21. ForwardRealToComplex01.java.

Does not implement the FFT algorithm. Implements a straight-forward sampled-data version of the continuous Fourier transform defined using integral calculus. See ForwardRealToComplexFFT01 for an FFT algorithm.

Returns real, imag, magnitude, and phase angle in degrees.

Incoming parameters are:

- double[] data - incoming real data
- double[] realOut - outgoing real data
- double[] imagOut - outgoing imaginary data
- double[] angleOut - outgoing phase angle in degrees
- double[] magnitude - outgoing amplitude spectrum
- int zero - the index of the incoming data sample that represents zero time
- double lowF - Low freq limit as fraction of sampling frequency
- double highF - High freq limit as fraction of sampling frequency

The frequency increment is the difference between high and low limits divided by the length of the magnitude array

The magnitude is computed as the square root of the sum of the squares of the real and imaginary parts. This value is divided by the incoming data length, which is given by data.length.

Returns a number of points in the frequency domain equal to the incoming data length regardless of the high and low frequency

Listing 21. ForwardRealToComplex01.java.

```
limits.
*****/

public class ForwardRealToComplex01{

    public static void transform(
                                double[] data,
                                double[] realOut,
                                double[] imagOut,
                                double[] angleOut,
                                double[] magnitude,
                                int zero,
                                double lowF,
                                double highF){
        double pi = Math.PI;//for convenience
        int dataLen = data.length;
        double delF = (highF-lowF)/data.length;
        //Outer loop iterates on frequency
        // values.
        for(int i=0; i < dataLen;i++){
            double freq = lowF + i*delF;
            double real = 0.0;
            double imag = 0.0;
            double ang = 0.0;
            //Inner loop iterates on time-
            // series points.
            for(int j=0; j < dataLen; j++){
                real += data[j]*Math.cos(
                                2*pi*freq*(j-zero));
                imag += data[j]*Math.sin(
                                2*pi*freq*(j-zero));
            }//end inner loop
            realOut[i] = real/dataLen;
            imagOut[i] = imag/dataLen;
            magnitude[i] = (Math.sqrt(
                                real*real + imag*imag))/dataLen;
        }
    }
}
```

Listing 21. ForwardRealToComplex01.java.

```
//Calculate and return the phase
// angle in degrees.
if(imag == 0.0 && real == 0.0){ang = 0.0;}
else{ang = Math.atan(imag/real)*180.0/pi;}

if(real < 0.0 && imag == 0.0){ang = 180.0;}
else if(real < 0.0 && imag == -0.0){
    ang = -180.0;}
else if(real < 0.0 && imag > 0.0){
    ang += 180.0;}
else if(real < 0.0 && imag < 0.0){
    ang += -180.0;}

    angleOut[i] = ang;
} //end outer loop
} //end transform method

} //end class ForwardRealToComplex01
```

Listing 22. Dsp030.java.

```
/* File Dsp030.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04
```

Uses an FFT algorithm to compute and display the magnitude of the spectral content for up to five sinusoids having different frequencies and amplitudes.

Listing 22. Dsp030.java.

See the program named Dsp028 for a program that does not use an FFT algorithm.

Gets input parameters from a file named Dsp030.txt. If that file doesn't exist in the current directory, the program uses a set of default parameters.

Each parameter value must be stored as characters on a separate line in the file named Dsp030.txt. The required parameters are as follows:

Data length as type int
Number of spectra as type int. Max value is 5.
List of sinusoid frequency values as type double.
List of sinusoid amplitude values as type double.

CAUTION: THE DATA LENGTH MUST BE A POWER OF TWO. OTHERWISE, THIS PROGRAM WILL FAIL TO RUN PROPERLY.

The number of values in each of the lists must match the value for the number of spectra.

Note: All frequency values are specified as a double representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is half the sampling frequency.

Here is a set of sample parameter values. Don't allow blank lines at the end of the data in the file.

128.0

Listing 22. Dsp030.java.

```
5
0.1
0.2
0.3
0.4
0.45
60
70
80
90
100
```

The plotting program that is used to plot the output data from this program requires that the program implement GraphIntfc01. For example, the plotting program named Graph03 can be used to plot the data produced by this program. When it is used, the usage information is:

```
java Graph03 Dsp030
```

A static method named transform belonging to the class named ForwardRealToComplexFFT01 is used to perform the actual spectral analysis. The method named transform implements an FFT algorithm. The FFT algorithm requires that the data length be a power of two.

Tested using SDK 1.4.2 under WinXP.

```
*****/
import java.util.*;
import java.io.*;

class Dsp030 implements GraphIntfc01{
    final double pi = Math.PI;//for simplification
```

Listing 22. Dsp030.java.

```
//Begin default parameters
int len = 128;//data length
int numberSpectra = 5;
//Frequencies of the sinusoids
double[] freq = {0.1,0.2,0.3,0.4,0.5};
//Amplitudes of the sinusoids
double[] amp = {60,70,80,90,100};
//End default parameters

//Following arrays will contain data that is
// input to the spectral analysis process.
double[] data1;
double[] data2;
double[] data3;
double[] data4;
double[] data5;

//Following arrays receive information back
// from the spectral analysis that is not used
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] magnitude1;
double[] magnitude2;
double[] magnitude3;
double[] magnitude4;
double[] magnitude5;

public Dsp030(){//constructor

    //Get the parameters from a file named
```

Listing 22. Dsp030.java.

```
// Dsp030.txt. Use the default parameters
// if the file doesn't exist in the current
// directory.
if(new File("Dsp030.txt").exists()){
    getParameters();
} //end if

//Note that this program always processes
// five sinusoids, even if fewer than five
// were requested as the input parameter
// for numberSpectra. In that case, the
// extras are processed using default values
// and simply ignored when the results are
// plotted.

//Create the raw data. Note that the
// argument for a sinusoid at half the
// sampling frequency would be (2*pi*x*0.5).
// This would represent one half cycle or pi
// radians per sample.
//First create empty array objects.
double[] data1 = new double[len];
double[] data2 = new double[len];
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];
//Now populate the array objects
for(int n = 0;n < len;n++){
    data1[n] = amp[0]*Math.cos(2*pi*n*freq[0]);
    data2[n] = amp[1]*Math.cos(2*pi*n*freq[1]);
    data3[n] = amp[2]*Math.cos(2*pi*n*freq[2]);
    data4[n] = amp[3]*Math.cos(2*pi*n*freq[3]);
    data5[n] = amp[4]*Math.cos(2*pi*n*freq[4]);
} //end for loop
//Compute magnitude spectra of the raw data
// and save it in output arrays. Note that
```

Listing 22. Dsp030.java.

```
// the real, imag, and angle arrays are not
// used later, so they are discarded each
// time a new spectral analysis is performed.
magnitude1 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplexFFT01.transform(data1,
                                     real, imag, angle, magnitude1);

magnitude2 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplexFFT01.transform(data2,
                                     real, imag, angle, magnitude2);

magnitude3 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplexFFT01.transform(data3,
                                     real, imag, angle, magnitude3);

magnitude4 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplexFFT01.transform(data4,
                                     real, imag, angle, magnitude4);

magnitude5 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplexFFT01.transform(data5,
```

Listing 22. Dsp030.java.

```
                                real,imag,angle,magnitude5);
} //end constructor
//-----//

//This method gets processing parameters from
// a file named Dsp030.txt and stores those
// parameters in instance variables belonging
// to the object of type Dsp030.
void getParameters(){
    int cnt = 0;
    //Temporary holding area for strings. Allow
    // space for a few blank lines at the end
    // of the data in the file.
    String[] data = new String[20];
    try{
        //Open an input stream.
        BufferedReader inData =
            new BufferedReader(new FileReader(
                                "Dsp030.txt"));
        //Read and save the strings from each of
        // the lines in the file. Be careful to
        // avoid having blank lines at the end,
        // which may cause an ArrayIndexOutOfBoundsException
        // exception to be thrown.
        while((data[cnt] =
                                inData.readLine()) != null){
            cnt++;
        } //end while
        inData.close();
    } catch(IOException e){}

    //Move the parameter values from the
    // temporary holding array into the instance
    // variables, converting from characters to
    // numeric values in the process.
    cnt = 0;
```

Listing 22. Dsp030.java.

```
len = (int)Double.parseDouble(data[cnt++]);
numberSpectra = (int)Double.parseDouble(
                                data[cnt++]);
for(int fCnt = 0;fCnt < numberSpectra;
    fCnt++){
    freq[fCnt] = Double.parseDouble(
                                data[cnt++]);
} //end for loop
for(int aCnt = 0;aCnt < numberSpectra;
    aCnt++){
    amp[aCnt] = Double.parseDouble(
                                data[cnt++]);
} //end for loop

//Print parameter values.
System.out.println();
System.out.println("Data length: " + len);
System.out.println(
    "Number spectra: " + numberSpectra);
System.out.println("Frequencies");
for(cnt = 0;cnt < numberSpectra;cnt++){
    System.out.println(freq[cnt]);
} //end for loop
System.out.println("Amplitudes");
for(cnt = 0;cnt < numberSpectra;cnt++){
    System.out.println(amp[cnt]);
} //end for loop

} //end getParameters
//-----//
//The following six methods are required by the
// interface named GraphIntfc01. The plotting
// program pulls the data values to be plotted
// by calling these methods.
public int getNmbr(){
    //Return number of functions to
```

Listing 22. Dsp030.java.

```
// process. Must not exceed 5.
return numberSpectra;
} //end getNnbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude1.length-1){
        return 0;
    }else{
        return magnitude1[index];
    } //end else
} //end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude2.length-1){
        return 0;
    }else{
        return magnitude2[index];
    } //end else
} //end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude3.length-1){
        return 0;
    }else{
        return magnitude3[index];
    } //end else
} //end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
```


Listing 22. Dsp030.java.

```
        if(index < 0 ||
            index > magnitude4.length-1){
            return 0;
        }else{
            return magnitude4[index];
        }//end else
    }//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude5.length-1){
        return 0;
    }else{
        return magnitude5[index];
    }//end else
}//end function
//-----//

}//end class Dsp030
```

Listing 23. ForwardRealToComplexFFT01.java.

```
/*File ForwardRealToComplexFFT01.java
Copyright 2004, R.G.Baldwin
Rev 5/14/04
```

The static method named transform performs a real to complex Fourier transform using a crippled

Listing 23. ForwardRealToComplexFFT01.java.

complex-to-complex FFT algorithm. It is crippled in the sense that it is not being used to its full potential as a complex-to-complex forward or inverse FFT algorithm.

See ForwardRealToComplex01 for a slower but more general approach that does not use an FFT algorithm.

Returns real, imag, magnitude, and phase angle in degrees.

Incoming parameters are:

- double[] data - incoming real data
- double[] realOut - outgoing real data
- double[] imagOut - outgoing imaginary data
- double[] angleOut - outgoing phase angle in degrees
- double[] magnitude - outgoing amplitude spectrum

The magnitude is computed as the square root of the sum of the squares of the real and imaginary parts. This value is divided by the incoming data length, which is given by data.length.

CAUTION: THE INCOMING DATA LENGTH MUST BE A POWER OF TWO. OTHERWISE, THIS PROGRAM WILL FAIL TO RUN PROPERLY.

Returns a number of points in the frequency domain equal to the incoming data length. Those points are uniformly distributed between zero and one less than the sampling frequency.

*****/

Listing 23. ForwardRealToComplexFFT01.java.

```
public class ForwardRealToComplexFFT01{

    public static void transform(
                                double[] data,
                                double[] realOut,
                                double[] imagOut,
                                double[] angleOut,
                                double[] magnitude){

        double pi = Math.PI;//for convenience
        int dataLen = data.length;
        //The complexToComplex FFT method does an
        // in-place transform causing the output
        // complex data to be stored in the arrays
        // containing the input complex data.
        // Therefore, it is necessary to copy the
        // input data to this method into the real
        // part of the complex data passed to the
        // complexToComplex method.
        System.arraycopy(data,0,realOut,0,dataLen);

        //Perform the spectral analysis. The results
        // are stored in realOut and imagOut. The +1
        // causes it to be a forward transform. A -1
        // would cause it to be an inverse transform.
        complexToComplex(1,dataLen,realOut,imagOut);

        //Compute the magnitude and the phase angle
        // in degrees.
        for(int cnt = 0;cnt < dataLen;cnt++){
            magnitude[cnt] =
                (Math.sqrt(realOut[cnt]*realOut[cnt]
                + imagOut[cnt]*imagOut[cnt]))/dataLen;

            if(imagOut[cnt] == 0.0
                && realOut[cnt] == 0.0){
                angleOut[cnt] = 0.0;
            }
        }
    }
}
```

Listing 23. ForwardRealToComplexFFT01.java.

```
        }//end if
        else{
            angleOut[cnt] = Math.atan(
                imagOut[cnt]/realOut[cnt])*180.0/pi;
        }//end else

        if(realOut[cnt] < 0.0
            && imagOut[cnt] == 0.0){
            angleOut[cnt] = 180.0;
        }else if(realOut[cnt] < 0.0
            && imagOut[cnt] == -0.0){
            angleOut[cnt] = -180.0;
        }else if(realOut[cnt] < 0.0
            && imagOut[cnt] > 0.0){
            angleOut[cnt] += 180.0;
        }else if(realOut[cnt] < 0.0
            && imagOut[cnt] < 0.0){
            angleOut[cnt] += -180.0;
        }//end else

    }//end for loop

} //end transform method
//-----//

//This method computes a complex-to-complex
// FFT. The value of sign must be 1 for a
// forward FFT.
public static void complexToComplex(
    int sign,
    int len,
    double real[],
    double imag[]){

    double scale = 1.0;
    //Reorder the input data into reverse binary
    // order.
```

Listing 23. ForwardRealToComplexFFT01.java.

```
int i,j;
for (i=j=0; i < len; ++i) {
    if (j>=i) {
        double tempr = real[j]*scale;
        double tempi = imag[j]*scale;
        real[j] = real[i]*scale;
        imag[j] = imag[i]*scale;
        real[i] = tempr;
        imag[i] = tempi;
    }//end if
    int m = len/2;
    while (m>=1 && j>=m) {
        j -= m;
        m /= 2;
    }//end while loop
    j += m;
} //end for loop

//Input data has been reordered.
int stage = 0;
int maxSpectraForStage,stepSize;
//Loop once for each stage in the spectral
// recombination process.
for(maxSpectraForStage = 1,
    stepSize = 2*maxSpectraForStage;
    maxSpectraForStage < len;
    maxSpectraForStage = stepSize,
    stepSize = 2*maxSpectraForStage){
    double deltaAngle =
        sign*Math.PI/maxSpectraForStage;
    //Loop once for each individual spectra
    for (int spectraCnt = 0;
        spectraCnt < maxSpectraForStage;
        ++spectraCnt){
        double angle = spectraCnt*deltaAngle;
        double realCorrection = Math.cos(angle);
```

Listing 23. ForwardRealToComplexFFT01.java.

```
double imagCorrection = Math.sin(angle);

int right = 0;
for (int left = spectraCnt;
     left < len; left += stepSize){
    right = left + maxSpectraForStage;
    double tempReal =
        realCorrection*real[right]
        - imagCorrection*imag[right];
    double tempImag =
        realCorrection*imag[right]
        + imagCorrection*real[right];
    real[right] = real[left]-tempReal;
    imag[right] = imag[left]-tempImag;
    real[left] += tempReal;
    imag[left] += tempImag;
} //end for loop
} //end for loop for individual spectra
maxSpectraForStage = stepSize;
} //end for loop for stages
} //end complexToComplex method

} //end class ForwardRealToComplexFFT01
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1482-Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
- File: Java1482.htm

- Published: 07/06/04

Baldwin explains several different programs used for spectral analysis. He also explains the impact of the sampling frequency and the Nyquist folding frequency on spectral analysis.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation ::: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1483-Spectrum Analysis using Java, Frequency Resolution versus Data Length

Baldwin provides the code and explains the requirements for using spectral analysis to resolve spectral peaks for pulses containing closely spaced truncated sinusoids.

Revised: Fri Oct 16 23:18:32 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Five pulses in the time domain](#)
 - [The lengths of the pulses](#)
 - [The program named Dsp031a](#)
 - [Time series containing sinusoidal pulses](#)
 - [Beginning of the class named Dsp031a](#)
 - [Spectral analysis results](#)

- [The program named Dsp031](#)
 - [Description](#)
 - [Uses a DFT algorithm](#)
 - [Beginning of the class named Dsp031](#)
 - [Perform the spectral analysis](#)
- [Separating closely spaced frequencies](#)
 - [Five new pulses](#)
- [The program named Dsp032a](#)
 - [Spectral analysis output](#)
- [The program named Dsp032](#)
 - [One more experiment](#)
 - [The five pulses](#)
 - [Spectral analysis results](#)
- [The program named Dsp033](#)
 - [The spectral analysis](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

The how and the why of spectral analysis

A previous module titled [Fun with Java, How and Why Spectral Analysis Works](#) explained some of the fundamentals regarding spectral analysis. An understanding of that module is a prerequisite to an understanding of this module.

Another previous module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#) presented and explained several Java programs for doing spectral analysis. In that module, I used a DFT program to illustrate several aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

I also used and briefly explained two different plotting programs that were originally explained in the earlier module titled [Plotting Engineering and Scientific Data using Java](#).

An understanding of the module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#) is also a prerequisite to an understanding of this module.

Frequency resolution versus data length

In this module I will use similar programs to explain and illustrate the manner in which spectral frequency resolution behaves with respect to data length.

A hypothetical situation

Consider a hypothetical situation in which you are performing spectral analysis on underwater acoustic signals in an attempt to identify enemy submarines.

You are aware that the enemy submarine contains a device that operates occasionally in short bursts. You are also aware that this device contains two rotating machines that rotate at almost but not quite the same speed.

During an operating burst of the device, each of the two machines contained in the device will emit acoustic energy that may appear as a peak in your spectral analysis output. (*Note that I said, "may appear" and did not say, "will appear."*) If you can identify the two peaks, you can conclusively identify the acoustic source as an enemy submarine.

The big question

How long must the operating bursts of this device be in order for you to resolve the peaks and identify the enemy submarine under ideal conditions?

That is the question that I will attempt to answer in this module by teaching you about the relationship between frequency resolution and data length.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Five pulses in the time domain.
- [Figure 2.](#) Spectral analyses of five pulses.
- [Figure 3.](#) Expanded spectral analyses of five pulses.
- [Figure 4.](#) Five pulses with two sinusoids each.
- [Figure 5.](#) Spectral analyses of five pulses.
- [Figure 6.](#) Five pulses with additive sinusoids.
- [Figure 7.](#) Spectral analyses of five pulses.
- [Figure 8.](#) Expanded spectral analyses of five pulses.

Listings

- [Listing 1.](#) Beginning of the class named Dsp031a.
- [Listing 2.](#) The constructor.
- [Listing 3.](#) Beginning of the class named Dsp031.
- [Listing 4.](#) Beginning of the constructor.
- [Listing 5.](#) Perform the spectral analysis.
- [Listing 6.](#) New code in the the program named Dsp032a.
- [Listing 7.](#) Computation of the frequencies.
- [Listing 8.](#) Create the pulses.
- [Listing 9.](#) Dsp031a.java.
- [Listing 10.](#) Dsp031.java.
- [Listing 11.](#) Dsp032a.java.
- [Listing 12.](#) File Dsp032.java.

- [Listing 13.](#) Dsp033a.java.
- [Listing 14.](#) File Dsp033.java.

Preview

Before I get into the technical details, here is a preview of the programs and their purposes that I will present and explain in this module:

- Dsp031 - Illustrates frequency resolution versus pulse length for pulses consisting of a truncated single sinusoid.
- Dsp031a - Displays the pulses analyzed by Dsp031.
- Dsp032 - Illustrates frequency resolution versus pulse length for pulses consisting of the sum of two truncated sinusoids with closely spaced frequencies.
- Dsp032a - Displays the pulses analyzed by Dsp032.
- Dsp033 - Illustrates frequency resolution versus pulse length for pulses consisting of the sum of two truncated sinusoids whose frequencies are barely resolvable.
- Dsp033a - Displays the pulses analyzed by Dsp033.

In addition, I will use the following programs that I explained in the module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#).

- ForwardRealToComplex01 - Class that implements the DFT algorithm for spectral analysis.
- Graph03 - Used to display various types of data. (*The concepts were explained in an earlier module.*)
- Graph06 - Also used to display various types of data in a somewhat different format. (*The concepts were also explained in an earlier module.*)

Discussion and sample code

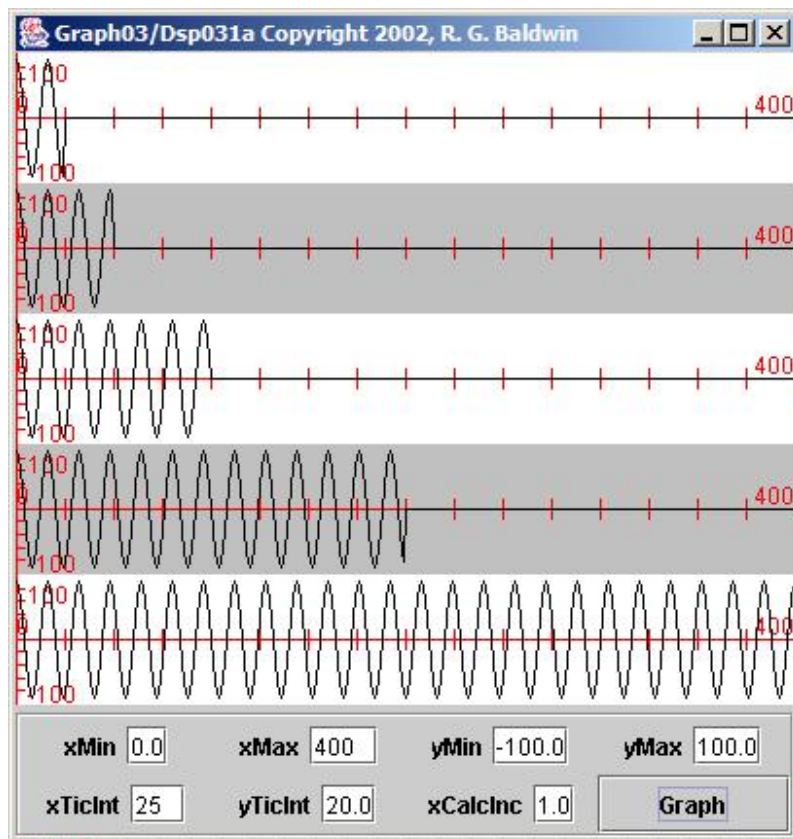
Five pulses in the time domain

Let's begin by looking at the time series data that will be used as input to the first spectral analysis experiment. [Figure 1](#) shows five pulses in the time

domain. [Figure 2](#) and [Figure 3](#) show the result of performing a spectral analysis on each of these pulses.

(The display in [Figure 1](#) was produced by the program named Dsp031a, which I will explain later.)

Figure 1. Five pulses in the time domain.



The lengths of the pulses

If you examine [Figure 1](#) carefully, you will see that each pulse is twice as long as the pulse above it. (*There is a tick mark on the horizontal axes every twenty-five samples.*) The bottom pulse is 400 samples long while the top pulse is 25 samples long.

Truncated sinusoids

Each pulse consists of a cosine wave that has been truncated at a different length. The frequency of the cosine wave is the same for every pulse. As you will see when we examine the code, the frequency of the cosine wave is 0.0625 times the sampling frequency. If you do the arithmetic, you will conclude that this results in 16 samples per cycle of the cosine wave.

In all five cases, the length of the time series upon which spectral analysis will be performed is 400 samples. For those four cases where the length of the pulse is less than 400 samples, the remaining samples in the time series have a value of zero.

Will compute at 400 frequencies

When the spectral analysis is performed later, the number of individual frequencies at which the amplitude of the spectral energy will be computed will be equal to the total data length. Therefore, the amplitude of the spectral energy will be computed at the same 400 frequencies for each of the five time series. That makes it convenient for us to stack the spectral plots up vertically and compare them (*as in [Figure 2](#)*). This makes it easy for us to compare the distribution of energy across the frequency spectrum for pulses of different lengths.

Graph03 and Graph06

The plots in [Figure 1](#) were produced using the program named **Graph03**. Other plots in this module will be produced using the program named **Graph06**. I explained those programs in earlier modules, and I provided the

source code for both programs in the previous module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#). Therefore, I won't repeat those explanations or provide the source code for those programs in this module.

The program named Dsp031a

A complete listing of the program named **Dsp031a** is provided in [Listing 9](#) near the end of the module.

This program displays sinusoidal pulses identical to those processed by the program named **Dsp031**, which will be discussed later.

Time series containing sinusoidal pulses

The program named **Dsp031a** creates and displays five separate time series, each 400 samples in length. Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of a truncated sinusoid. The frequency of the sinusoid for each of the pulses is the same.

Frequency values are specified as type **double** as a fraction of the sampling frequency. The frequency of each sinusoid is 0.0625 times the sampling frequency.

The pulse lengths

The lengths of the five pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

Beginning of the class named Dsp031a

This program is very similar to programs that I explained in previous modules in this series, so my explanation will be very brief. As usual, I will explain the program in fragments.

The beginning of the class, along with the declaration and initialization of several variables is shown in [Listing 1](#). The names of the variables along with the embedded comments should make the code self explanatory.

Listing 1. Beginning of the class named Dsp031a.

```
class Dsp031a implements GraphIntfc01{
    final double pi = Math.PI;

    int len = 400;//data length
    int numberPulses = 5;
    //Frequency of the sinusoids
    double freq = 0.0625;
    //Amplitude of the sinusoids
    double amp = 160;

    //Following arrays will contain sinusoidal
    data
    double[] data1 = new double[len];
    double[] data2 = new double[len];
    double[] data3 = new double[len];
    double[] data4 = new double[len];
    double[] data5 = new double[len];
```


The constructor

[Listing 2](#) shows the constructor, which creates the raw sinusoidal data and stores that data in the array objects created in [Listing 1](#).

(Recall that all element values in the array objects are initialized with a value of zero. Therefore, the code in [Listing 2](#) only needs to store the non-zero values in the array objects.)

Listing 2. The constructor.

Listing 2. The constructor.

```
public Dsp031a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/8;x++){
        data2[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/4;x++){
        data3[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len;x++){
        data5[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

} //end constructor
```

The code in the conditional clause of each of the for loops in [Listing 2](#) controls the length of each of the sinusoidal pulses.

The interface methods

As you can see in [Listing 1](#), the class implements the interface named **GraphIntfc01**. I introduced this interface in the earlier module titled [Plotting](#)

[Engineering and Scientific Data using Java](#) and also discussed it in the previous module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#).

The remaining code for the class named **Dsp031a** consists of the methods necessary to satisfy the interface. These methods are called by the plotting programs named **Graph03** and **Graph06** to obtain and plot the data returned by the methods. As implemented in **Dsp031a**, these interface methods return the values stored in the array objects referred to by **data1** through **data5**. Thus, the values stored in those array objects are plotted in [Figure 1](#).

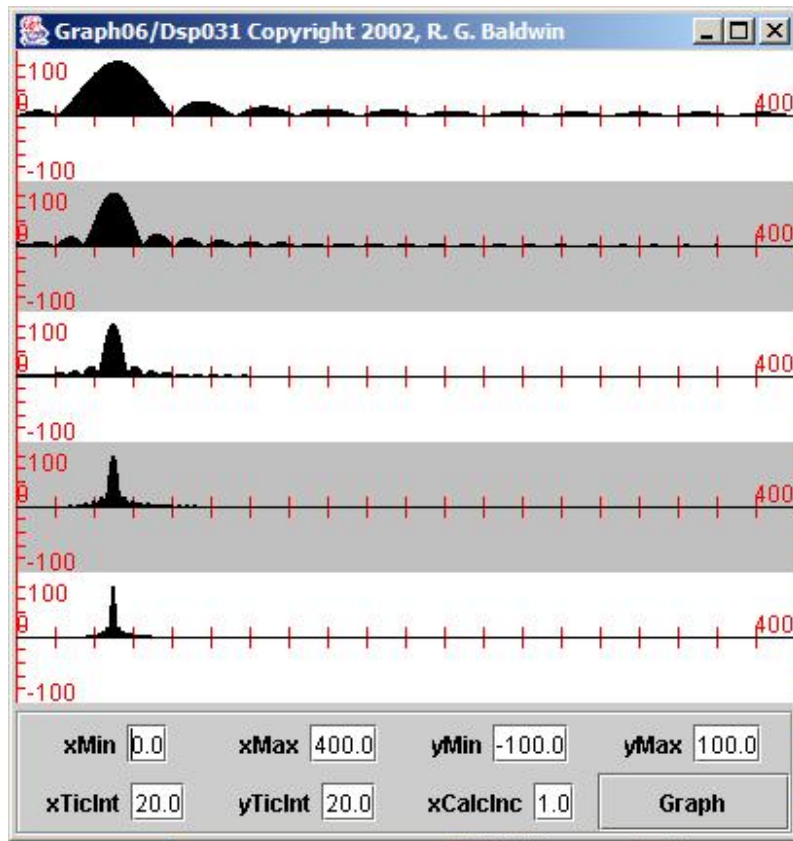
Spectral analysis results

[Figure 2](#) shows the result of using the program named **Dsp031** to perform a spectral analysis on each of the five pulses shown in [Figure 1](#). These results were plotted using the program named **Graph06**. With this plotting program, each data value is plotted as a vertical bar. However, in this case, the sides of each of the bars are so close together that the area under the spectral curve appears to be solid black.

(When you run this program, you can expand the display to full screen and see the individual vertical bars. However, I can't do that and maintain the narrow publication format required for this module.)

Figure 2. Spectral analyses of five pulses.

Figure 2. Spectral analyses of five pulses.



Interpretation of the results

Before I get into the interpretation, I need to point out that I normalized the data plotted in [Figure 2](#) to cause each spectral peak to have approximately the same value. Otherwise, the spectral analysis result values for the short pulses would have been too small to be visible in this plotting format.

Therefore, the fact that the area under the curve in the top plot is greater than the area under the curve in the bottom plot doesn't indicate that the first pulse contains more energy than the last pulse. It simply means that I normalized the data for best results in plotting.

Spectrum of an ideal sinusoid

That having been said, different people will probably interpret these results in different ways. Let's begin by stating that the theoretical spectrum for a sinusoid of infinite length in the absence of noise is a single vertical line having zero width and infinite height.

In the real world of measurements, however, there is no such thing as a sinusoid of infinite length. Rather, every measurement that we make must truncate the sinusoid at some point in time. For a theoretical signal of infinite length, every spectral analysis that we can perform is an imperfect estimate of the spectrum.

Two viewpoints

There are at least two ways to think of the pulses shown in [Figure 1](#).

1. Each pulse is a truncated section of an ideal sinusoid of infinite length.
2. Each pulse is a signal having a definite planned start and stop time.

The way that you interpret the results shown in [Figure 2](#) depends on your viewpoint regarding the pulses.

The first viewpoint

If your viewpoint is that each pulse is a truncated section of an ideal sinusoid of infinite length, then the width of each of the peaks (*beyond zero width*) is the result of measurement error introduced by the truncation process.

The second viewpoint

If your viewpoint is that each pulse is a signal having a definite planned start and stop time, then the widths and the shape of each of the peaks describes the full range of frequency components required to physically generate such a pulse. This is the viewpoint that is consistent with the [hypothetical situation](#) involving a device on a submarine that I described earlier in this module.

A simplified hypothetical situation

Assume for the moment that the hypothetical device on the submarine contains only one rotating machine and that this device is turned on and off occasionally in short bursts. Because of the rotating machine, when the device is turned on, it will emit acoustic energy whose frequency matches the rotating speed of the machine.

(In reality, it will probably also emit acoustic energy at other frequencies as well, but we will consider it to be a very ideal machine. We will also assume the complete absence of any other acoustic noise in the environment.)

Assume that you have a recording window of 400 samples, and that you are able to record five such bursts within each of five separate recording windows. Further assume that the lengths of the individual bursts match the time periods indicated by the pulses in [Figure 1](#).

The spectra of the bursts

If you perform spectral analysis on each of the five individual 400-sample windows containing the bursts, and if you normalize the peak values for plotting purposes, you should get results similar to those shown in [Figure 2](#).

The spectral bandwidth of the signal

The frequency range over which energy is distributed is referred to as the bandwidth of the signal. As you can see in [Figure 2](#), shorter pulses require wider bandwidth.

For example, considerably more bandwidth is required of a communication system that is required to reliably transmit a series of short truncated sinusoids than one that is only required to reliably transmit a continuous tone at a single frequency.

At the same time, it is very difficult to convey very much information with a signal consisting of a continuous tone at a single frequency (*other than the fact that the tone exists*) . Communication systems designed to convey information usually encode that information by either turning the tone on and off or by causing it to shift among a set of previously defined frequencies. The tone is often referred to as the *carrier* and the encoding of the information is often referred to as *modulating* the carrier.

Thus, you need greater bandwidth to reliably convey more information.

The relationship between pulse length and bandwidth

So far, we can draw one important conclusion from our experiment.

Shorter pulses require greater bandwidth.

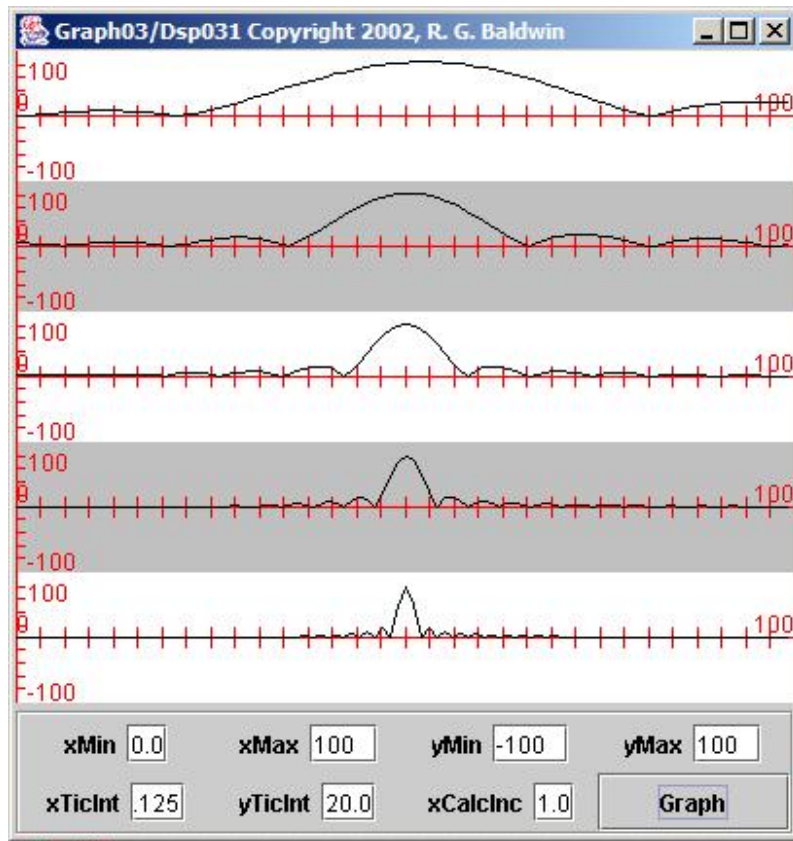
This leads to an important question. What is the numerical relationship between pulse length and bandwidth? Although we can draw the above general conclusion from [Figure 2](#), it is hard to draw any quantitative conclusions from [Figure 2](#). That brings us to the expanded plot of the spectral data shown in [Figure 3](#).

An expanded plot of the spectral results

[Figure 3](#) shows the left one-fourth of the spectral results from [Figure 2](#) plotted in the same horizontal space. In other words, [Figure 3](#) discards the upper three-fourths of the spectral results from [Figure 2](#) and shows only the lower one-fourth of the spectral results on an expanded scale. [Figure 3](#) also provides tick marks that make it convenient to perform measurements on the plots.

Also, whereas [Figure 2](#) was plotted using the program named **Graph06** , [Figure 3](#) was plotted using the program named **Graph03** . Thus, [Figure 3](#) uses a different plotting format than [Figure 2](#)

Figure 3. Expanded spectral analyses of five pulses.



Picking numeric values

The curves in [Figure 3](#) are spread out to the point that we can pick some approximate numeric values off the plot, and from this, we can draw a very significant conclusion.

For purposes of our approximation, consider the bandwidth to be the distance along the frequency axis between the points where the curves touch zero on either side of the peak. Using this approximation, the bandwidth indicated by the spectral analyses in [Figure 3](#) shows the bandwidth of each spectrum to be twice as wide as the one below it.

Referring back to [Figure 1](#), recall that the length of each pulse was half that of the one below it. The conclusion is:

The bandwidth of a truncated sinusoidal pulse is inversely proportional to the length of the pulse.

If you reduce the length of the pulse by a factor of two, you must double the bandwidth of a transmission system designed to reliably transmit a pulse of that length.

This will also be an important conclusion regarding our ability to separate and identify the two spectral peaks in the burst of acoustic energy described in our original [hypothetical situation](#).

The program named Dsp031

The generation of the signals and the spectral analysis for the results presented in [Figure 2](#) and [Figure 3](#) were performed using the program named **Dsp031**. A complete listing of the program is shown in [Listing 10](#) near the end of the module.

Description

This program performs spectral analyses on five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths. (*The lengths of the individual pulses match that shown in [Figure 1](#).*) Each pulse consists of a truncated sinusoid. The frequency of the sinusoid for each pulse is the same.

All frequency values are specified as type **double** as a fraction of the sampling frequency. The frequency of all five sinusoids is 0.0625 times the

sampling frequency.

The lengths of the pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

If this sounds familiar, it is because the pulses are identical to those displayed in [Figure 1](#) and discussed under **Dsp031a** above.

Uses a DFT algorithm

The spectral analysis process uses a DFT algorithm and computes the amplitude of the spectral energy at 400 equally spaced frequencies between zero and the folding frequency.

(Recall from the module titled [Spectrum Analysis using Java](#), [Sampling Frequency](#), [Folding Frequency](#), and the [FFT Algorithm](#) that the folding frequency is one-half the sampling frequency.)

This program computes and displays the amplitude spectrum at frequency intervals that are one-half of the frequency intervals for a typical FFT algorithm.

Normalize the results

The results of the spectral analysis are multiplied by the reciprocal of the lengths of the individual pulses to normalize all five plots to the same peak value. Otherwise, the results for the short pulses would be too small to see on the plots.

Will discuss in fragments

Once again, this program is very similar to programs explained in the previous module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#). Therefore, this discussion will be very brief.

Beginning of the class named Dsp031

The code in [Listing 3](#) declares and initializes some variables and creates the array objects that will contain the sinusoidal pulses.

In addition, the code in [Listing 3](#) declares reference variables that will be used to refer to array objects containing results of the spectral analysis process that are not used in this program.

Finally, [Listing 3](#) declares reference variables that will be used to refer to array objects containing the results plotted in [Figure 2](#) and [Figure 3](#).

Given the names of the variables, the comments, and what you learned in the earlier modules, the code in [Listing 3](#) should be self explanatory.

Listing 3. Beginning of the class named Dsp031.

```
class Dsp031 implements GraphIntf01{
    final double pi = Math.PI;

    int len = 400;//data length
    //Sample that represents zero time.
    int zeroTime = 0;
    //Low and high frequency limits for the
    // spectral analysis.
```

Listing 3. Beginning of the class named Dsp031.

```
double lowF = 0.0;
double highF = 0.5;
int numberSpectra = 5;
//Frequency of the sinusoids
double freq = 0.0625;
//Amplitude of the sinusoids
double amp = 160;

//Following arrays will contain data that is
// input to the spectral analysis process.
double[] data1 = new double[len];
double[] data2 = new double[len];
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];

//Following arrays receive information back
// from the spectral analysis that is not used
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] mag1;
double[] mag2;
double[] mag3;
double[] mag4;
double[] mag5;
```

The constructor begins in [Listing 4](#). The code in [Listing 4](#) is identical to that shown earlier in [Listing 2](#). This code generates the five sinusoidal pulses and stores the data representing those pulses in the arrays referred to by **data1** through **data5**. So far, except for the declaration of some extra variables, this program isn't much different from the program named **Dsp031a** discussed earlier in this module.

Listing 4. Beginning of the constructor.

```
public Dsp031(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/8;x++){
        data2[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/4;x++){
        data3[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len;x++){
        data5[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop
```

Perform the spectral analysis

The remainder of the constructor is shown in [Listing 5](#). This code calls the **transform** method of the **ForwardRealToComplex01** class five times in succession to perform the spectral analysis on each of the five pulses shown in [Figure 1](#).

*(I explained the **transform** method in detail in the previous module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#).)*

Listing 5. Perform the spectral analysis.

```
mag1 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data1, real,
    imag, angle, mag1, zeroTime, lowF, highF);

mag2 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data2, real,
    imag, angle, mag2, zeroTime, lowF, highF);

mag3 = new double[len];
real = new double[len];
imag = new double[len];
```

Listing 5. Perform the spectral analysis.

```
angle = new double[len];
ForwardRealToComplex01.transform(data3, real,
    imag, angle, mag3, zeroTime, lowF, highF);

mag4 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data4, real,
    imag, angle, mag4, zeroTime, lowF, highF);

mag5 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data5, real,
    imag, angle, mag5, zeroTime, lowF, highF);

} //end constructor
```

Each time the transform method is called, it computes the magnitude spectrum for the incoming data and saves it in the output array.

(Note that the real, imag, and angle arrays are not used later, so they are discarded each time a new spectral analysis is performed.)

The interface methods

The **Dsp031** class also implements the interface named **GraphIntfc01**. The remaining code in the program consists of the methods required to satisfy that interface. Except for the identification of the arrays from which the methods

extract data to be returned for plotting, these methods are identical to those defined in the earlier class named **Dsp031a** . Therefore, I won't discuss them further.

What we have learned so far

So far, the main things that we have learned is that shorter pulses require greater bandwidth, and the bandwidth required to faithfully represent a truncated sinusoidal pulse is the reciprocal of the length of the pulse.

Separating closely spaced frequencies

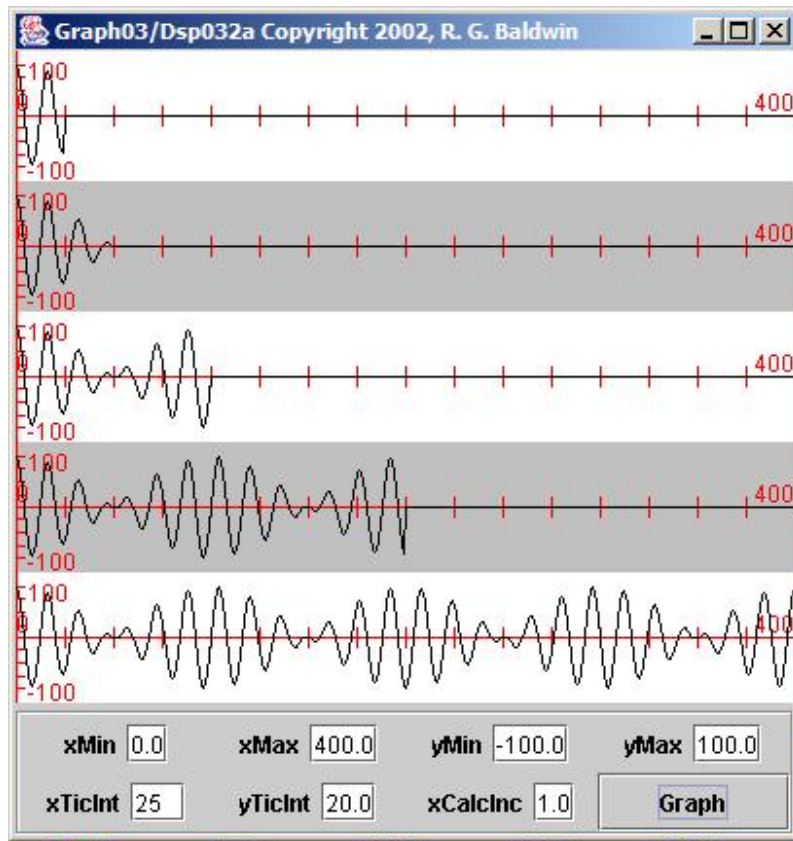
Now we will look at the issues involved in using spectral analysis to separate and identify the frequencies of two closely-spaced spectral peaks for a pulse composed of the sum of two sinusoids. Once again, we will begin by looking at some results and then discuss the code that produced those results.

Five new pulses

The five pulses that we will be working with in this example are shown in [Figure 4](#). As you can see, these pulses are a little more ugly than the pulses shown in [Figure 1](#). As you can also see, as was the case in [Figure 1](#), each pulse appears to be a shorter or longer version of the other pulses in terms of its waveform.

Figure 4. Five pulses with two sinusoids each.

Figure 4. Five pulses with two sinusoids each.



Produced by Dsp032a

The plots in [Figure 4](#) were produced by the program named **Dsp032a**, which I will briefly discuss later. (A complete listing of the program is shown in [Listing 11](#) near the end of the module.) This program creates and displays pulses identical to those processed by the program named **Dsp032**, which I will also briefly discuss later. (A complete listing of the program named **Dsp32** is presented in [Listing 12](#).)

Five time series containing pulses

The program creates and displays five separate time series, each 400 samples in length. Each time series contains a pulse and the pulses are different lengths. As before, each of the pulses shown in [Figure 4](#) is half the length of the pulse below it.

The sum of two sinusoids

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids for all pulses are the same.

All frequency values are specified as type **double** as a fraction of the sampling frequency. The frequencies of the two sinusoids are equidistant from a center frequency of 0.0625 times the sampling frequency.

*(Recall that 0.0625 was the frequency of the only sinusoid contained in the pulses shown in [Figure 1](#) and processed by the program named **Dsp031** .)*

The frequencies and pulse lengths

The frequency of one sinusoid is $(0.0625 - 2.0/\text{len})$ times the sampling frequency, where len is the length of the time series containing the pulse. *(The value for len is 400 samples in this program.)* The frequency of the other sinusoid is $(0.0625 + 2.0/\text{len})$ times the sampling frequency.

The lengths of the five pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

(Note that [Figure 4](#) has tick marks every 25 samples.)

The program named Dsp032a

The only new code in this program is the code in the constructor that creates the pulses and stores them in the data arrays. This code consists of five separate **for** loops, one for each pulse. The code for the first **for** loop, which is typical of the five, is shown in [Listing 6](#).

Listing 6. New code in the the program named Dsp032a.

```
public Dsp032a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq1)
                +
    amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    //... code removed for brevity

    }//end constructor
```

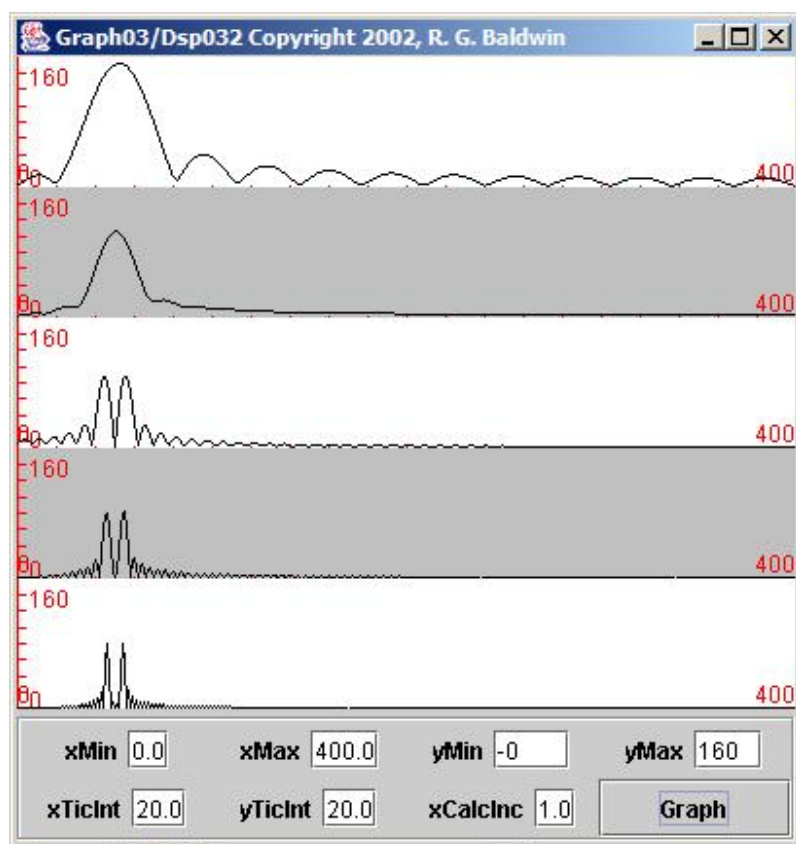
As you can see from [Listing 6](#), the values that make up the pulse are produced by adding together the values of two different cosine functions having different frequencies. The values for **freq1** and **freq2** are as described above.

You can view the remainder of this program in [Listing 11](#).

Spectral analysis output

The results of running the program named **Dsp032** and displaying the results with the program named **Graph03** are shown in [Figure 5](#).

Figure 5. Spectral analyses of five pulses.



Each of the peaks in the third, fourth, and fifth plots in [Figure 5](#) corresponds to the frequency of one of the two sinusoids that were added together to produce the pulses shown in [Figure 4](#).

Can we answer the original question now?

The question posed in the original [hypothetical situation](#) was "how long must the operating bursts of this device be in order for you to resolve the peaks and identify the enemy submarine under ideal conditions?"

We are looking at very ideal conditions in [Figure 4](#) and [Figure 5](#). In particular, the pulses exist completely in the absence of noise.

(The existence of wide-band noise added to the pulses in [Figure 4](#) would cause a change in the spectral results in [Figure 5](#). That change might be described as having the appearance of grass and weeds growing on the baseline across the entire spectrum. The stronger the wide-band noise, the taller would be the weeds.)

Cannot resolve two peaks for first two pulses

Clearly for the ideal condition of recording the bursts in the total absence of noise, you cannot resolve the peaks from the top two plots in [Figure 5](#). For those two pulses, the spectral peaks simply merge together to form a single broad peak. Therefore, for this amount of separation between the frequencies of the two sinusoids, the lengths of the first two pulses in [Figure 4](#) are insufficient to allow for separation and identification of the separate peaks.

We must be in luck

We seem to have the problem bracketed. (*Were we really lucky, or did I plan it this way?*) Under the ideal conditions of this experiment, the peaks are separable in the middle plot of [Figure 5](#). Thus, for the amount of separation between the frequencies of the two sinusoids, the length of the third pulse in [Figure 4](#) is sufficient to allow for separation and identification of the separate peaks.

A qualified answer to the question

The peaks are even better separated in the bottom two plots in [Figure 5](#). For the five pulses used in this experiment and the amount of separation between the frequencies of the two sinusoids, any pulse as long or longer than the length of the third pulse is [Figure 4](#) is sufficient to allow for separation and identification of the separate peaks.

What about the effects of noise?

If you were to add a nominal amount of wide-band noise to the mix, it would become more difficult to resolve the peaks for the bottom three plots in [Figure 5](#) because the peaks would be growing out of a bed of weeds.

(If you add enough wide-band noise, you couldn't resolve the peaks using any of the plots, because the peaks would be completely "lost in the noise.")

What can we learn from this?

Since we have concluded that the middle pulse in [Figure 4](#) is sufficiently long to allow us to resolve the two peaks, let's see what we can learn from the parameters that describe that pulse.

Pulse length and frequency separation

To begin with, the length of the pulse is 100 samples.

What about the frequency separation of the two sinusoids? Recall that the frequency of one sinusoid is $(0.0625 - 2.0/\text{len})$ times the sampling frequency, where len is the length of the time series containing the pulse. The frequency of the other sinusoid is $(0.0625 + 2.0/\text{len})$ times the sampling frequency.

Thus, total separation between the two frequencies is $4/\text{len}$, or $4/400$. Dividing through by 4 we see that the separation between the two frequencies is $1/100$.

Eureka, we have found it

For the third pulse, the frequency separation **is the reciprocal of the length of the pulse** . Also, the length of the third pulse is barely sufficient to allow for separation and identification of the two peaks in the spectrum.

Thus, the two spectral peaks are separable in the absence of noise if the frequency separation is the reciprocal of the pulse length. *(That is too good to be a coincidence. I must have planned that way.)*

Thus, we have reached another conclusion. Under ideal conditions, the two peaks in the spectrum can be resolved when the separation between the frequencies of the two sinusoids is equal to the reciprocal of the pulse length.

The general answer

There is no single answer to the question *"how long must the operating bursts of this device be in order for you to resolve the peaks and identify the enemy submarine under ideal conditions?"*

The answer depends on the frequency separation. The general answer is that the length of the bursts must be at least as long as the reciprocal of the frequency separation for the two sinusoids. If the separation is large, the pulse length may be short. If the separation is small, the pulse length must be long.

The program named Dsp032

As I indicated earlier, the plots shown in [Figure 5](#) were the result of running the program named **Dsp032** and displaying the data with the program named **Graph03** .

The only thing that is new in this program is the code that generates the five pulses and saves them in their respective data arrays. Even that code is not

really new, because it is identical to the code shown in [Listing 6](#). Therefore, I won't discuss this program further in this module.

One more experiment

As you can surmise from the conclusions reached above, in order to be able to resolve the two peaks in the spectrum, you can either keep the pulse length the same and increase the frequency separation, or you can keep the frequency separation the same and increase the pulse length.

Let's examine an example where we keep the pulse lengths the same as before and adjust the frequency separation between the two sinusoids to make it barely possible to resolve the peaks for each of the five pulses.

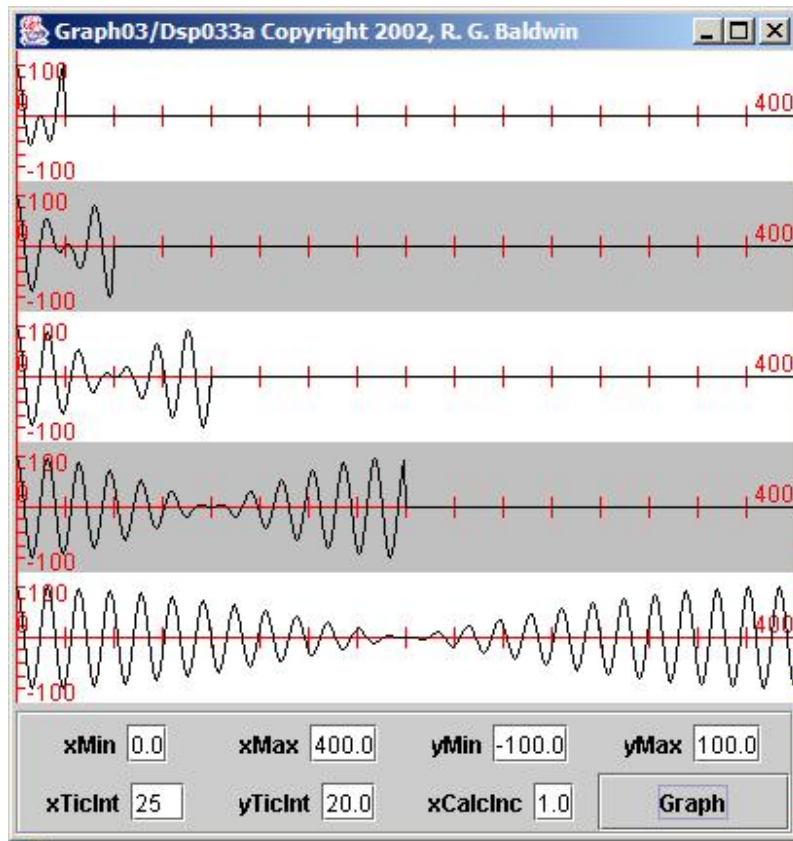
We will need to increase the frequency separation for the first two pulses, and we can decrease the frequency separation for the fourth and fifth pulses. We will leave the frequency separation the same as before for the third pulse since it already seems to have the optimum relationship between pulse length and frequency separation.

The five pulses

The five pulses used in this experiment are shown in [Figure 6](#).

Figure 6. Five pulses with additive sinusoids.

Figure 6. Five pulses with additive sinusoids.



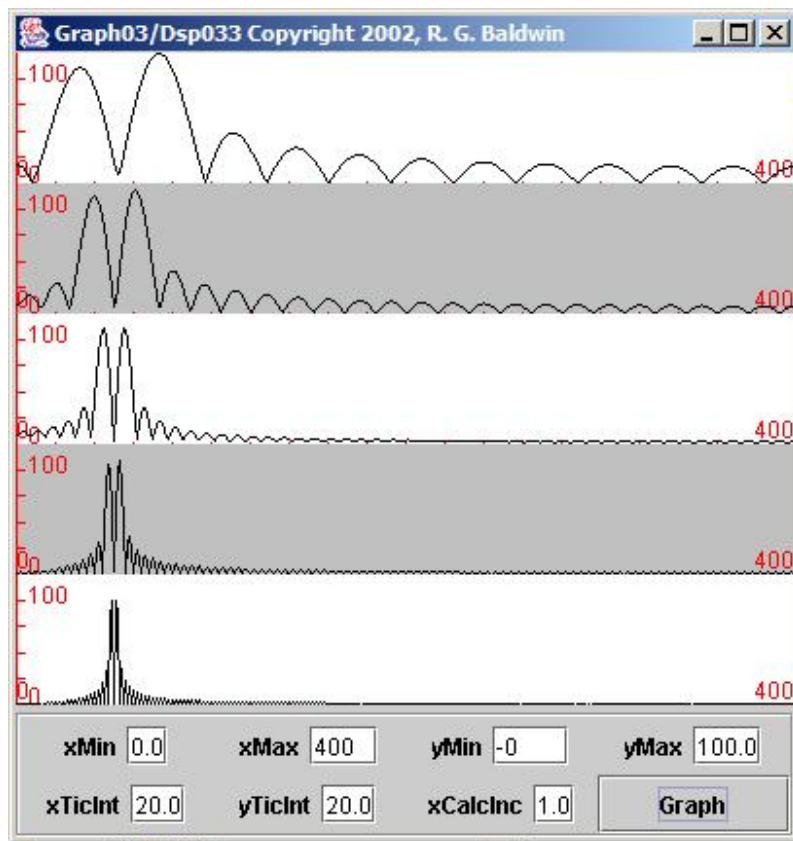
Unlike in the previous two cases shown in [Figure 1](#) and [Figure 4](#), each of these pulses has a different shape from the others. In other words, in the previous two cases, each pulse simply looked like a longer or shorter version of the other pulses. That is not the case in this example.

(Note however that the third pulse in [Figure 6](#) looks just like the third pulse in [Figure 4](#). They were created using the same parameters. However, none of the other pulses in [Figure 6](#) look like the corresponding pulses in [Figure 4](#), and none of the pulses in [Figure 6](#) look like the pulses in [Figure 1](#).)

Spectral analysis results

[Figure 7](#) shows the result of performing spectral analysis on the five time series containing the pulses shown in [Figure 6](#).

Figure 7. Spectral analyses of five pulses.



Peaks for first two pulses are now resolvable

When we examine the code, you will see that the frequency separation for the first two pulses has been increased to the reciprocal of the pulse length in each

case. This results in the two peaks in the spectrum for each of the first two pulses being resolvable in [Figure 7](#).

Third pulse hasn't changed

The spectrum for the third pulse shown in [Figure 7](#) is almost identical to the spectrum for the third pulse shown in [Figure 5](#). The only difference is that I had to decrease the vertical scaling on all of the plots in [Figure 5](#) to keep the peak in the top plot within the bounds of the plot.

Spectral peaks for last two pulses are closer

When we examine the code, you will also see that the frequency separation for the last two pulses has been decreased to the reciprocal of the pulse length in each case. This results in the two peaks in the spectrum for each of the last two pulses being closer than before in [Figure 7](#).

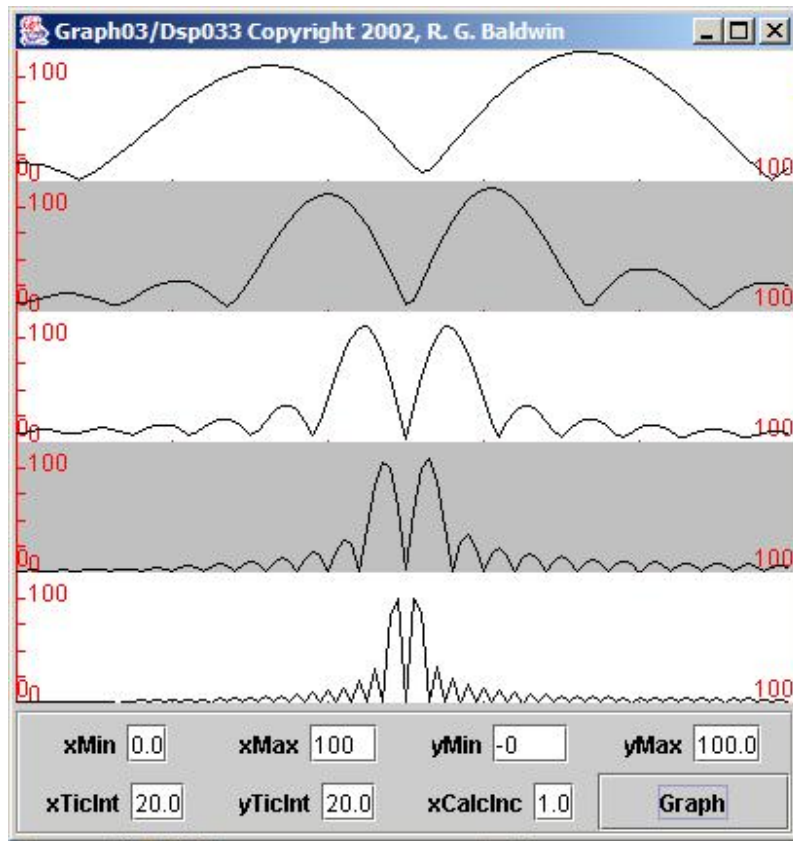
The peaks in the bottom two plots in [Figure 7](#) appear to be resolvable, but we can't be absolutely certain because they are so close together, particularly for the last plot.

(If you expand the Frame to full screen when you run this program, you will see that the two peaks are resolvable, but I can't do that and stay within this narrow publication format.)

Expand the horizontal plotting scale

[Figure 8](#) adjusts the plotting parameters to cause the left-most one-fourth of the data in [Figure 7](#) to be plotted in the full width of the Frame in [Figure 8](#).

Figure 8. Expanded spectral analyses of five pulses.



The peaks are barely resolvable

[Figure 8](#) shows that the two peaks are barely resolvable for all five of the pulses shown in [Figure 6](#).

(There is no space between the peaks at the baseline in [Figure 8](#), but the plots do go almost down to the baseline half way between the two peaks.)

The program named Dsp033

The plots in [Figure 7](#) and [Figure 8](#) were produced by running the program named **Dsp033** and plotting the results with the program named **Graph03** .

A complete listing of the program named **Dsp033** is shown in [Listing 14](#) near the end of the module.

This program is the same as **Dsp032** except that the separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse in each case.

The program performs spectral analysis on five separate time series, each 400 samples in length. Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids are equidistant from a center frequency of 0.0625 times the sampling frequency. The total separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse.

All frequency values are specified as type **double** as a fraction of the sampling frequency.

The lengths of the pulses are:

- 25 samples
- 50 samples
- 100 samples
- 200 samples
- 400 samples

The spectral analysis

The spectral analysis computes the spectra at 400 equally spaced frequencies between zero and the folding frequency (*one-half the sampling frequency*) .

The results of the spectral analysis are multiplied by the reciprocal of the lengths of the individual pulses to normalize the five plots. Otherwise, the results for the short pulses would be too small to see on the plots.

Because of the similarity of this program to the previous programs, my discussion of the code will be very brief.

Computation of the frequencies

The code in [Listing 7](#) shows the computation of the frequencies of the sinusoids that will be added together to form each of the five pulses.

Listing 7. Computation of the frequencies.

```
//Frequencies of the sinusoids
double freq1a = 0.0625 - 8.0/len;
double freq2a = 0.0625 + 8.0/len;

double freq1b = 0.0625 - 4.0/len;
double freq2b = 0.0625 + 4.0/len;

double freq1c = 0.0625 - 2.0/len;
double freq2c = 0.0625 + 2.0/len;

double freq1d = 0.0625 - 1.0/len;
double freq2d = 0.0625 + 1.0/len;

double freq1e = 0.0625 - 0.5/len;
double freq2e = 0.0625 + 0.5/len;
```

Create the pulses

The code in [Listing 8](#) uses those frequency values to create the data for the pulses and to store that data in the arrays used to hold the pulses.

Listing 8. Create the pulses.

Listing 8. Create the pulses.

```
//Create the raw data
for(int x = 0;x < len/16;x++){
    data1[x] = amp*Math.cos(2*pi*x*freq1a)
              +
amp*Math.cos(2*pi*x*freq2a);
} //end for loop

for(int x = 0;x < len/8;x++){
    data2[x] = amp*Math.cos(2*pi*x*freq1b)
              +
amp*Math.cos(2*pi*x*freq2b);
} //end for loop

for(int x = 0;x < len/4;x++){
    data3[x] = amp*Math.cos(2*pi*x*freq1c)
              +
amp*Math.cos(2*pi*x*freq2c);
} //end for loop

for(int x = 0;x < len/2;x++){
    data4[x] = amp*Math.cos(2*pi*x*freq1d)
              +
amp*Math.cos(2*pi*x*freq2d);
} //end for loop

for(int x = 0;x < len;x++){
    data5[x] = amp*Math.cos(2*pi*x*freq1e)
              +
amp*Math.cos(2*pi*x*freq2e);
} //end for loop
```

Other than the code shown in [Listing 7](#) and [Listing 8](#), the program named Dsp033 is the same as the programs that were previously explained, and I

won't discuss it further.

Run the programs

I encourage you to copy, compile, and run the programs provided in this module. Experiment with them, making changes and observing the results of your changes.

Create more complex experiments. For example, you could create pulses containing three or more sinusoids at closely spaced frequencies, and you could cause the amplitudes of the sinusoids to be different. See what it takes to cause the peaks in the spectra of those pulses to be separable and identifiable.

If you really want to get fancy, you could create a pulse consisting of a sinusoid whose frequency changes with time from the beginning to the end of the pulse. (*A pulse of this type is often referred to as a frequency modulated sweep signal.*) See what you can conclude from doing spectral analysis on a pulse of this type.

Try using the random number generator of the **Math** class to add some random noise to every value in the 400-sample time series. See what this does to your spectral analysis results.

Move the center frequency up and down the frequency axis. See if you can explain what happens as the center frequency approaches zero and as the center frequency approaches the folding frequency.

Most of all, enjoy yourself and learn something in the process.

Summary

This program provides the code for three spectral analysis experiments of increasing complexity.

Bandwidth versus pulse length

The first experiment performs spectral analyses on five simple pulses consisting of truncated sinusoids. This experiment shows:

- Shorter pulses require greater bandwidth.
- The bandwidth of a truncated sinusoidal pulse is inversely proportional to the length of the pulse.

Peak resolution versus pulse length and frequency separation

The second experiment performs spectral analyses on five more complex pulses consisting of the sum of two truncated sinusoids having closely spaced frequencies. The purpose is to determine the required length of the pulse in order to use spectral analysis to resolve spectral peaks attributable to the two sinusoids. The experiment shows that the peaks are barely resolvable when the length of the pulse is the reciprocal of the frequency separation between the two sinusoids.

Five pulses with barely resolvable spectral peaks

The third experiment also performs spectral analyses on five pulses consisting of the sum of two truncated sinusoids having closely spaced frequencies. In this case, the frequency separation for each pulse is the reciprocal of the length of the pulse. The results of the spectral analysis reinforce the conclusions drawn in the second experiment.

What's next?

So far, the modules in this series have ignored the complex nature of the results of spectral analysis. The complex results have been converted into real results by computing the square root of the sum of the squares of the real and imaginary parts.

The next module in the series will meet the issue of complex spectral results head on and will explain the concept of phase angle. In addition, the module will explain the behavior of the phase angle with respect to time shifts in the input time series.

Complete program listings

Complete listings of the main programs discussed in this module are provided in this section. Listings for other programs mentioned in the module, such as **Graph03** and **Graph06**, are provided in other modules. Those modules are identified in the text of this module.

Listing 9. Dsp031a.java.

```
/* File Dsp031a.java  
Copyright 2004, R.G.Baldwin  
Revised 5/17/2004
```

Displays sinusoidal pulses identical to those processed by Dsp031.

Creates and displays five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of a truncated sinusoid. The frequency of the sinusoid for all pulses is the same.

All frequency values are specified as type double as a fraction of the sampling frequency.

The frequency of all sinusoids is 0.0625 times the sampling frequency.

The lengths of the pulses are:

Listing 9. Dsp031a.java.

25 samples
50 samples
100 samples
200 samples
400 samples

Tested using J2SEE 1.4.2 under WinXP.

```
*****/  
import java.util.*;  
  
class Dsp031a implements GraphIntfc01{  
    final double pi = Math.PI;  
  
    int len = 400;//data length  
    int numberPulses = 5;  
    //Frequency of the sinusoids  
    double freq = 0.0625;  
    //Amplitude of the sinusoids  
    double amp = 160;  
  
    //Following arrays will contain sinusoidal data  
    double[] data1 = new double[len];  
    double[] data2 = new double[len];  
    double[] data3 = new double[len];  
    double[] data4 = new double[len];  
    double[] data5 = new double[len];  
  
    public Dsp031a(){//constructor  
  
        //Create the raw data  
        for(int x = 0;x < len/16;x++){  
            data1[x] = amp*Math.cos(2*pi*x*freq);  
        }//end for loop  
  
        for(int x = 0;x < len/8;x++){  
            data2[x] = amp*Math.cos(2*pi*x*freq);
```

Listing 9. Dsp031a.java.

```
    }//end for loop

    for(int x = 0;x < len/4;x++){
        data3[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len;x++){
        data5[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

} //end constructor

//-----//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data1.length-1){
        return 0;
    }else{
        //Scale the amplitude of the pulses to make
        // them compatible with the default
        // plotting amplitude of 100.0.
        return data1[index]*90.0/amp;
    } //end else
} //end function
```

Listing 9. Dsp031a.java.

```
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data2.length-1){
        return 0;
    }else{
        return data2[index]*90.0/amp;
    }//end else
}//end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data3.length-1){
        return 0;
    }else{
        return data3[index]*90.0/amp;
    }//end else
}//end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data4.length-1){
        return 0;
    }else{
        return data4[index]*90.0/amp;
    }//end else
}//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data5.length-1){
        return 0;
    }else{
        return data5[index]*90.0/amp;
    }//end else
}//end function
```

Listing 9. Dsp031a.java.

```
}//end sample class Dsp031a
```

Listing 10. Dsp031.java.

```
/* File Dsp031.java  
Copyright 2004, R.G.Baldwin  
Revised 5/17/2004
```

Performs spectral analysis on five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of a truncated sinusoid. The frequency of the sinusoid for all pulses is the same.

All frequency values are specified as type double as a fraction of the sampling frequency.

The frequency of all sinusoids is 0.0625 times the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples

Listing 10. Dsp031.java.

100 samples
200 samples
400 samples

The spectral analysis computes the spectra at 400 equally spaced points between zero and the folding frequency (one-half the sampling frequency).

The results of the spectral analysis are multiplied by the reciprocal of the lengths of the individual pulses to normalize all five plots to the same peak value. Otherwise, the results for the short pulses would be too small to see on the plots.

Tested using J2SEE 1.4.2 under WinXP.

```
*****/  
import java.util.*;  
  
class Dsp031 implements GraphIntf01{  
    final double pi = Math.PI;  
  
    int len = 400;//data length  
    //Sample that represents zero time.  
    int zeroTime = 0;  
    //Low and high frequency limits for the  
    // spectral analysis.  
    double lowF = 0.0;  
    double highF = 0.5;  
    int numberSpectra = 5;  
    //Frequency of the sinusoids  
    double freq = 0.0625;  
    //Amplitude of the sinusoids  
    double amp = 160;
```


Listing 10. Dsp031.java.

```
//Following arrays will contain data that is
// input to the spectral analysis process.
double[] data1 = new double[len];
double[] data2 = new double[len];
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];

//Following arrays receive information back
// from the spectral analysis that is not used
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] mag1;
double[] mag2;
double[] mag3;
double[] mag4;
double[] mag5;

public Dsp031(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/8;x++){
        data2[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/4;x++){
```

Listing 10. Dsp031.java.

```
        data3[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    for(int x = 0;x < len;x++){
        data5[x] = amp*Math.cos(2*pi*x*freq);
    }//end for loop

    //Compute magnitude spectra of the raw data
    // and save it in output arrays. Note that
    // the real, imag, and angle arrays are not
    // used later, so they are discarded each
    // time a new spectral analysis is performed.
    mag1 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data1,real,
        imag,angle,mag1,zeroTime,lowF,highF);

    mag2 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data2,real,
        imag,angle,mag2,zeroTime,lowF,highF);

    mag3 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data3,real,
```

Listing 10. Dsp031.java.

```
        imag, angle, mag3, zeroTime, lowF, highF);

    mag4 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data4, real,
        imag, angle, mag4, zeroTime, lowF, highF);

    mag5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5, real,
        imag, angle, mag5, zeroTime, lowF, highF);

} //end constructor

//-----//
//The following six methods are required by the
// interface named GraphIntf01.
public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int) Math.round(x);
    if(index < 0 || index > mag1.length-1){
        return 0;
    } else{
        //Scale the magnitude data by the
        // reciprocal of the length of the sinusoid
        // to normalize the five plots to the same
        // peak value.
```

Listing 10. Dsp031.java.

```
        return mag1[index]*16.0;
    }//end else
} //end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag2.length-1){
        return 0;
    }else{
        return mag2[index]*8.0;
    }//end else
} //end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag3.length-1){
        return 0;
    }else{
        return mag3[index]*4.0;
    }//end else
} //end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag4.length-1){
        return 0;
    }else{
        return mag4[index]*2.0;
    }//end else
} //end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag5.length-1){
        return 0;
    }else{
```

Listing 10. Dsp031.java.

```
        return mag5[index]*1.0;
    }//end else
} //end function

} //end sample class Dsp031
```

Listing 11. Dsp032a.java.

```
/* File Dsp032a.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004
```

Displays sinusoidal pulses identical to those processed by Dsp032.

Creates and displays five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids for all pulses are the same.

All frequency values are specified as type double as a fraction of the sampling frequency.

The frequencies of the two sinusoids are

Listing 11. Dsp032a.java.

equidistant from 0.0625 times the sampling frequency.

The frequency of one sinusoid is $(0.0625 - 2.0/\text{len})$ times the sampling frequency.

The frequency of the other sinusoid is $(0.0625 + 2.0/\text{len})$ times the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

Tested using J2SEE 1.4.2 under WinXP.

```
*****/  
import java.util.*;
```

```
class Dsp032a implements GraphIntfc01{  
    final double pi = Math.PI;  
  
    int len = 400;//data length  
    int numberPulses = 5;  
    //Frequencies of the sinusoids  
    double freq1 = 0.0625 - 2.0/len;  
    double freq2 = 0.0625 + 2.0/len;  
  
    //Amplitude of the sinusoids  
    double amp = 160;  
  
    //Following arrays will contain sinusoidal data  
    double[] data1 = new double[len];  
    double[] data2 = new double[len];
```

Listing 11. Dsp032a.java.

```
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];

public Dsp032a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq1)
                + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/8;x++){
        data2[x] = amp*Math.cos(2*pi*x*freq1)
                + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/4;x++){
        data3[x] = amp*Math.cos(2*pi*x*freq1)
                + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq1)
                + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len;x++){
        data5[x] = amp*Math.cos(2*pi*x*freq1)
                + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

} //end constructor

//-----//
//The following six methods are required by the
```

Listing 11. Dsp032a.java.

```
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data1.length-1){
        return 0;
    }else{
        //Scale the amplitude of the pulses to make
        // them compatible with the default
        // plotting amplitude of 100.0.
        return data1[index]*40.0/amp;
    } //end else
} //end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data2.length-1){
        return 0;
    }else{
        return data2[index]*40.0/amp;
    } //end else
} //end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data3.length-1){
        return 0;
    }else{
        return data3[index]*40.0/amp;
    } //end else
} //end function
```


Listing 11. Dsp032a.java.

```
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data4.length-1){
        return 0;
    }else{
        return data4[index]*40.0/amp;
    }//end else
}//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data5.length-1){
        return 0;
    }else{
        return data5[index]*40.0/amp;
    }//end else
}//end function

}//end sample class Dsp032a
```

Listing 12. File Dsp032.java.

```
/* File Dsp032.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004
```

Performs spectral analysis on five separate time series, each 400 samples in length.

Listing 12. File Dsp032.java.

Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids for all pulses are the same.

All frequency values are specified as type double as a fraction of the sampling frequency.

The frequencies of the two sinusoids are equidistant from 0.0625 times the sampling frequency.

The frequency of one sinusoid is $(0.0625 - 2.0/\text{len})$ times the sampling frequency.

The frequency of the other sinusoid is $(0.0625 + 2.0/\text{len})$ times the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

The spectral analysis computes the spectra at 400 equally spaced points between zero and the folding frequency (one-half the sampling frequency).

The results of the spectral analysis are multiplied by the reciprocal of the lengths of

Listing 12. File Dsp032.java.

the individual pulses to normalize the five plots. Otherwise, the results for the short pulses would be too small to see on the plots.

Tested using J2SEE 1.4.2 under WinXP.

*****/

```
import java.util.*;
```

```
class Dsp032 implements GraphIntf01{  
    final double pi = Math.PI;
```

```
    int len = 400;//data length  
    //Sample that represents zero time.  
    int zeroTime = 0;  
    //Low and high frequency limits for the  
    // spectral analysis.  
    double lowF = 0.0;  
    double highF = 0.5;  
    int numberSpectra = 5;  
    //Frequencies of the sinusoids  
    double freq1 = 0.0625 - 2.0/len;  
    double freq2 = 0.0625 + 2.0/len;
```

```
    //Amplitude of the sinusoids  
    double amp = 160;
```

```
    //Following arrays will contain data that is  
    // input to the spectral analysis process.  
    double[] data1 = new double[len];  
    double[] data2 = new double[len];  
    double[] data3 = new double[len];  
    double[] data4 = new double[len];  
    double[] data5 = new double[len];
```

```
    //Following arrays receive information back  
    // from the spectral analysis that is not used
```

Listing 12. File Dsp032.java.

```
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] mag1;
double[] mag2;
double[] mag3;
double[] mag4;
double[] mag5;

public Dsp032(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/8;x++){
        data2[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/4;x++){
        data3[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq1)
                    + amp*Math.cos(2*pi*x*freq2);
    }//end for loop
```

Listing 12. File Dsp032.java.

```
for(int x = 0;x < len;x++){
    data5[x] = amp*Math.cos(2*pi*x*freq1)
              + amp*Math.cos(2*pi*x*freq2);
} //end for loop
```

```
//Compute magnitude spectra of the raw data
// and save it in output arrays. Note that
// the real, imag, and angle arrays are not
// used later, so they are discarded each
// time a new spectral analysis is performed.
```

```
mag1 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data1,real,
    imag,angle,mag1,zeroTime,lowF,highF);
```

```
mag2 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data2,real,
    imag,angle,mag2,zeroTime,lowF,highF);
```

```
mag3 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data3,real,
    imag,angle,mag3,zeroTime,lowF,highF);
```

```
mag4 = new double[len];
real = new double[len];
imag = new double[len];
```

Listing 12. File Dsp032.java.

```
    angle = new double[len];
    ForwardRealToComplex01.transform(data4, real,
        imag, angle, mag4, zeroTime, lowF, highF);

    mag5 = new double[len];
    real = new double[len];
    imag = new double[len];
    angle = new double[len];
    ForwardRealToComplex01.transform(data5, real,
        imag, angle, mag5, zeroTime, lowF, highF);

} //end constructor

//-----//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNnbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
} //end getNnbr
//-----//
public double f1(double x){
    int index = (int) Math.round(x);
    if(index < 0 || index > mag1.length-1){
        return 0;
    } else{
        //Scale the magnitude data by the
        // reciprocal of the length of the sinusoid
        // to normalize the five plots to the same
        // peak value.
        return mag1[index]*16.0;
    } //end else
} //end function
//-----//
public double f2(double x){
```

Listing 12. File Dsp032.java.

```
        int index = (int)Math.round(x);
        if(index < 0 || index > mag2.length-1){
            return 0;
        }else{
            return mag2[index]*8.0;
        }//end else
    }//end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag3.length-1){
        return 0;
    }else{
        return mag3[index]*4.0;
    }//end else
}//end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag4.length-1){
        return 0;
    }else{
        return mag4[index]*2.0;
    }//end else
}//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag5.length-1){
        return 0;
    }else{
        return mag5[index]*1.0;
    }//end else
}//end function

}//end sample class Dsp032
```

Listing 12. File Dsp032.java.

Listing 13. Dsp033a.java.

```
/* File Dsp033a.java  
Copyright 2004, R.G.Baldwin  
Revised 5/17/2004
```

Displays sinusoidal pulses identical to those processed by Dsp033.

Creates and displays five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids are equidistant from 0.0625 times the sampling frequency. The total separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse.

All frequency values are specified as type double as a fraction of the sampling frequency.

The lengths of the pulses are:

```
25 samples  
50 samples  
100 samples
```


Listing 13. Dsp033a.java.

200 samples

400 samples

Tested using J2SEE 1.4.2 under WinXP.

*****/

```
import java.util.*;
```

```
class Dsp033a implements GraphIntfc01{  
    final double pi = Math.PI;
```

```
    int len = 400;//data length
```

```
    int numberPulses = 5;
```

```
    //Frequencies of the sinusoids
```

```
    double freq1a = 0.0625 - 8.0/len;
```

```
    double freq2a = 0.0625 + 8.0/len;
```

```
    double freq1b = 0.0625 - 4.0/len;
```

```
    double freq2b = 0.0625 + 4.0/len;
```

```
    double freq1c = 0.0625 - 2.0/len;
```

```
    double freq2c = 0.0625 + 2.0/len;
```

```
    double freq1d = 0.0625 - 1.0/len;
```

```
    double freq2d = 0.0625 + 1.0/len;
```

```
    double freq1e = 0.0625 - 0.5/len;
```

```
    double freq2e = 0.0625 + 0.5/len;
```

```
    //Amplitude of the sinusoids
```

```
    double amp = 160;
```

```
    //Following arrays will contain sinusoidal data
```

```
    double[] data1 = new double[len];
```

```
    double[] data2 = new double[len];
```

```
    double[] data3 = new double[len];
```

```
    double[] data4 = new double[len];
```

Listing 13. Dsp033a.java.

```
double[] data5 = new double[len];

public Dsp033a(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq1a)
                +
amp*Math.cos(2*pi*x*freq2a);
    }//end for loop

    for(int x = 0;x < len/8;x++){
        data2[x] = amp*Math.cos(2*pi*x*freq1b)
                +
amp*Math.cos(2*pi*x*freq2b);
    }//end for loop

    for(int x = 0;x < len/4;x++){
        data3[x] = amp*Math.cos(2*pi*x*freq1c)
                +
amp*Math.cos(2*pi*x*freq2c);
    }//end for loop

    for(int x = 0;x < len/2;x++){
        data4[x] = amp*Math.cos(2*pi*x*freq1d)
                +
amp*Math.cos(2*pi*x*freq2d);
    }//end for loop

    for(int x = 0;x < len;x++){
        data5[x] = amp*Math.cos(2*pi*x*freq1e)
                + amp*Math.cos(2*pi*x*freq2e);
    }//end for loop

} //end constructor
```

Listing 13. Dsp033a.java.

```
//-----//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data1.length-1){
        return 0;
    }else{
        //Scale the amplitude of the pulses to make
        // them compatible with the default
        // plotting amplitude of 100.0.
        return data1[index]*40.0/amp;
    } //end else
} //end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data2.length-1){
        return 0;
    }else{
        return data2[index]*40.0/amp;
    } //end else
} //end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data3.length-1){
        return 0;
    }else{
        return data3[index]*40.0/amp;
```

Listing 13. Dsp033a.java.

```
        }//end else
    }//end function
    //-----//
    public double f4(double x){
        int index = (int)Math.round(x);
        if(index < 0 || index > data4.length-1){
            return 0;
        }else{
            return data4[index]*40.0/amp;
        }//end else
    }//end function
    //-----//
    public double f5(double x){
        int index = (int)Math.round(x);
        if(index < 0 || index > data5.length-1){
            return 0;
        }else{
            return data5[index]*40.0/amp;
        }//end else
    }//end function

} //end sample class Dsp033a
```

Listing 14. File Dsp033.java.

```
/* File Dsp033.java
Copyright 2004, R.G.Baldwin
Revised 5/17/2004
```

Listing 14. File Dsp033.java.

Same as Dsp032 except that the separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse.

Performs spectral analysis on five separate time series, each 400 samples in length.

Each time series contains a pulse and the pulses are different lengths.

Each pulse consists of the sum of two sinusoids at closely spaced frequencies. The frequencies of the two sinusoids are equidistant from 0.0625 times the sampling frequency. The total separation between the frequencies of the two sinusoids is the reciprocal of the length of the pulse.

All frequency values are specified as type double as a fraction of the sampling frequency.

The lengths of the pulses are:

25 samples
50 samples
100 samples
200 samples
400 samples

The spectral analysis computes the spectra at 400 equally spaced points between zero and the folding frequency (one-half the sampling frequency).

The results of the spectral analysis are multiplied by the reciprocal of the lengths of

Listing 14. File Dsp033.java.

the individual pulses to normalize the five plots. Otherwise, the results for the short pulses would be too small to see on the plots.

Tested using J2SEE 1.4.2 under WinXP.

*****/

```
import java.util.*;
```

```
class Dsp033 implements GraphIntf01{
    final double pi = Math.PI;
```

```

    int len = 400;//data length
    //Sample that represents zero time.
    int zeroTime = 0;
    //Low and high frequency limits for the
    // spectral analysis.
    double lowF = 0.0;
    double highF = 0.5;
    int numberSpectra = 5;
    //Frequencies of the sinusoids
    double freq1a = 0.0625 - 8.0/len;
    double freq2a = 0.0625 + 8.0/len;

    double freq1b = 0.0625 - 4.0/len;
    double freq2b = 0.0625 + 4.0/len;

    double freq1c = 0.0625 - 2.0/len;
    double freq2c = 0.0625 + 2.0/len;

    double freq1d = 0.0625 - 1.0/len;
    double freq2d = 0.0625 + 1.0/len;

    double freq1e = 0.0625 - 0.5/len;
    double freq2e = 0.0625 + 0.5/len;
```

Listing 14. File Dsp033.java.

```
//Amplitude of the sinusoids
double amp = 160;

//Following arrays will contain data that is
// input to the spectral analysis process.
double[] data1 = new double[len];
double[] data2 = new double[len];
double[] data3 = new double[len];
double[] data4 = new double[len];
double[] data5 = new double[len];

//Following arrays receive information back
// from the spectral analysis that is not used
// in this program.
double[] real;
double[] imag;
double[] angle;

//Following arrays receive the magnitude
// spectral information back from the spectral
// analysis process.
double[] mag1;
double[] mag2;
double[] mag3;
double[] mag4;
double[] mag5;

public Dsp033(){//constructor

    //Create the raw data
    for(int x = 0;x < len/16;x++){
        data1[x] = amp*Math.cos(2*pi*x*freq1a)
                +
        amp*Math.cos(2*pi*x*freq2a);
    }//end for loop
```

Listing 14. File Dsp033.java.

```
        for(int x = 0;x < len/8;x++){
            data2[x] = amp*Math.cos(2*pi*x*freq1b)
                    +
amp*Math.cos(2*pi*x*freq2b);
        }//end for loop

        for(int x = 0;x < len/4;x++){
            data3[x] = amp*Math.cos(2*pi*x*freq1c)
                    +
amp*Math.cos(2*pi*x*freq2c);
        }//end for loop

        for(int x = 0;x < len/2;x++){
            data4[x] = amp*Math.cos(2*pi*x*freq1d)
                    +
amp*Math.cos(2*pi*x*freq2d);
        }//end for loop

        for(int x = 0;x < len;x++){
            data5[x] = amp*Math.cos(2*pi*x*freq1e)
                    + amp*Math.cos(2*pi*x*freq2e);
        }//end for loop


        //Compute magnitude spectra of the raw data
        // and save it in output arrays. Note that
        // the real, imag, and angle arrays are not
        // used later, so they are discarded each
        // time a new spectral analysis is performed.
        mag1 = new double[len];
        real = new double[len];
        imag = new double[len];
        angle = new double[len];
        ForwardRealToComplex01.transform(data1,real,
            imag,angle,mag1,zeroTime,lowF,highF);
```


Listing 14. File Dsp033.java.

```
mag2 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data2, real,
    imag, angle, mag2, zeroTime, lowF, highF);

mag3 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data3, real,
    imag, angle, mag3, zeroTime, lowF, highF);

mag4 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data4, real,
    imag, angle, mag4, zeroTime, lowF, highF);

mag5 = new double[len];
real = new double[len];
imag = new double[len];
angle = new double[len];
ForwardRealToComplex01.transform(data5, real,
    imag, angle, mag5, zeroTime, lowF, highF);

} //end constructor

//-----//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
```

Listing 14. File Dsp033.java.

```
        return 5;
    }//end getNmbr
    //-----//
    public double f1(double x){
        int index = (int)Math.round(x);
        if(index < 0 || index > mag1.length-1){
            return 0;
        }else{
            //Scale the magnitude data by the
            // reciprocal of the length of the sinusoid
            // to normalize the five plots to the same
            // peak value.
            return mag1[index]*16.0;
        }//end else
    }//end function
    //-----//
    public double f2(double x){
        int index = (int)Math.round(x);
        if(index < 0 || index > mag2.length-1){
            return 0;
        }else{
            return mag2[index]*8.0;
        }//end else
    }//end function
    //-----//
    public double f3(double x){
        int index = (int)Math.round(x);
        if(index < 0 || index > mag3.length-1){
            return 0;
        }else{
            return mag3[index]*4.0;
        }//end else
    }//end function
    //-----//
    public double f4(double x){
        int index = (int)Math.round(x);
```

Listing 14. File Dsp033.java.

```
        if(index < 0 || index > mag4.length-1){
            return 0;
        }else{
            return mag4[index]*2.0;
        }//end else
    }//end function
    //-----//
    public double f5(double x){
        int index = (int)Math.round(x);
        if(index < 0 || index > mag5.length-1){
            return 0;
        }else{
            return mag5[index]*1.0;
        }//end else
    }//end function

} //end sample class Dsp033
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1483-Spectrum Analysis using Java, Frequency Resolution versus Data Length
- File: Java1483.htm
- Published: 08/10/04

Baldwin provides the code and explains the requirements for using spectral analysis to resolve spectral peaks for pulses containing closely spaced truncated sinusoids.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1484-Spectrum Analysis using Java, Complex Spectrum and Phase Angle
Baldwin discusses the complex spectrum and explains the relationship
between the phase angle and shifts in the time domain.

Revised: Fri Oct 16 18:22:28 CDT 2015

This page is included in the following books:

- [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The order of the plotted results](#)
 - [Parameter file format](#)
 - [Format for frequency specifications](#)
 - [The plotting programs](#)
 - [Spectral analysis](#)
 - [Results](#)
 - [Computing the Fourier transform](#)
 - [The program named Dsp034](#)
 - [The beginning of the Dsp034 class](#)

- [More examples](#)
 - [The simplest pulse of all, an impulse](#)
 - [Results of spectral analysis on an impulse](#)
 - [Introduce a time delay](#)
 - [Introduce a large time delay](#)
 - [A boxcar pulse](#)
 - [The magic of non real-time digital processing](#)
- [Run the program](#)
- [Summary](#)
- [What's next?](#)
- [Complete program listing](#)
- [Miscellaneous](#)

Preface

Spectral analysis

A previous module titled [Fun with Java, How and Why Spectral Analysis Works](#) explained some of the fundamentals regarding spectral analysis.

The module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#) presented and explained several Java programs for doing spectral analysis, including both DFT programs and FFT programs. That module illustrated the fundamental aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

The module titled [Spectrum Analysis using Java, Frequency Resolution versus Data Length](#) used similar Java programs to explain spectral frequency resolution.

In this module, I will deal with issues involving the complex spectrum, the phase angle, and shifts in the time domain.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Sample parameters for testing.
- [Figure 2.](#) Spectral analysis of a damped pulse.
- [Figure 3.](#) Fourier transform equations.
- [Figure 4.](#) The simplest pulse of all, an impulse.
- [Figure 5.](#) Spectral analysis of an impulse at zero time.
- [Figure 6.](#) Introduce a time delay.
- [Figure 7.](#) Spectral analysis of an impulse with a one-sample delay.
- [Figure 8.](#) Introduce a large time delay.
- [Figure 9.](#) Spectral analysis of impulse with five-sample delay.
- [Figure 10.](#) A boxcar pulse.
- [Figure 11.](#) Spectral analysis of 11-sample boxcar pulse.
- [Figure 12.](#) Shift the time base.
- [Figure 13.](#) Spectral analysis of 11-sample boxcar centered in time.

Listings

- [Listing 1.](#) Computation of the phase angle.
- [Listing 2.](#) The beginning of the Dsp034 class.
- [Listing 3.](#) Beginning of the constructor.
- [Listing 4.](#) Store the pulse in the time series.
- [Listing 5.](#) Perform the spectral analysis.
- [Listing 6.](#) Dsp034.java.

Preview

In this module, I will present and explain a program named **Dsp034** . This program will be used to illustrate the behavior of the complex spectrum and the phase angle for several different scenarios.

In addition, I will use the following programs that I explained in the module titled [Spectrum Analysis using Java](#), [Sampling Frequency](#), [Folding Frequency](#), and the [FFT Algorithm](#).

- ForwardRealToComplex01 - Class that implements the DFT algorithm for spectral analysis.
- Graph03 - Used to display various types of data. (*The concepts were explained in an earlier module.*)
- Graph06 - Also used to display various types of data in a somewhat different format. (*The concepts were also explained in an earlier module.*)

Discussion and sample code

The program named **Dsp034** , when run in conjunction with either **Graph03** or **Graph06** computes and plots the amplitude, real, imaginary, and phase angle spectrum for a pulse that is read from a file named **Dsp034.txt** . If that file doesn't exist in the current directory, the program uses a set of default parameters that describe a damped sinusoidal pulse. The program also plots the pulse itself in addition to the spectral analysis results listed above.

The order of the plotted results

When the data is plotted (see [Figure 2](#)) using the programs **Graph03** or **Graph06** , the order of the plots from top to bottom in the display is:

- The pulse
- The amplitude spectrum
- The real spectrum
- The imaginary spectrum
- The phase angle in degrees

Parameter file format

Each parameter value must be stored as characters on a separate line in the file named **Dsp034.txt** . The required parameters and their order and type are as follows:

- Data length as type **int**
- Pulse length as type **int**
- Sample number representing zero time as type **int**
- Low frequency bound as type **double**
- High frequency bound as type **double**
- Sample values for the pulse as type **double** or **int** (*they are automatically converted to type **double** if they are provided as type **int***)

The number of sample values for the pulse must match the value for the pulse length.

Format for frequency specifications

All frequency values are specified as a **double** representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is half the sampling frequency.

Sample parameters for testing

[Figure 1](#) provides a set of sample parameter values that can be used to test the program. This sample data describes a triangular pulse. Be careful when you create the file containing these values. Don't allow blank lines at the end of the data in the file.

Figure 1. Sample parameters for testing.

Figure 1. Sample parameters for testing.

```
400
11
0
0.0
0.5
0
0
0
45
90
135
90
45
0
0
0
```

The plotting programs

The plotting program that is used to plot the output data from this program requires that the program implement **GraphIntfc01** .

(I explained the plotting programs and this interface in earlier modules.)

For example, the plotting program named **Graph03** can be used to plot the data produced by this program. When it is used, the following should be entered at the command-line prompt:

java Graph03 Dsp034

Spectral analysis

A static method named **transform** belonging to the class named **ForwardRealToComplex01** is used to perform the actual spectral analysis.

*(I explained this class and the **transform** method in the earlier module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#). However, I skipped over that portion of the method that computes the phase angle. I will explain that portion of the **transform** method in this module.)*

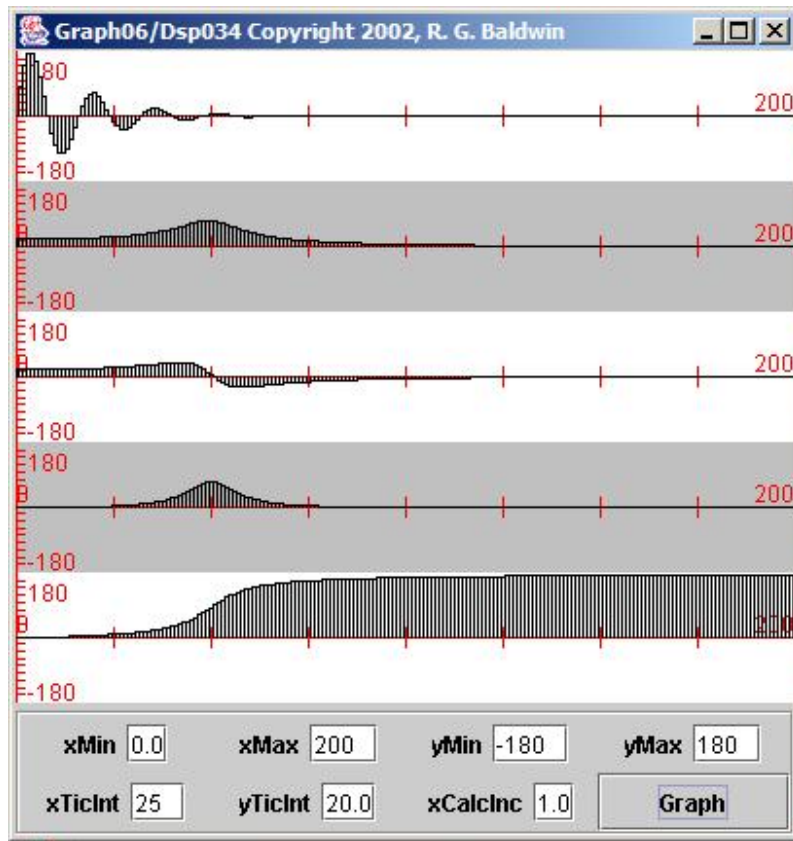
The method named **transform** does not implement an FFT algorithm. Rather, it implements a DFT algorithm, which is more general than, but much slower than an FFT algorithm. (See the program named **Dsp030** in the module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#) for the use of an FFT algorithm.)

Results

Before getting into the technical details of the program, let's take a look at some results. [Figure 2](#) shows the results produced by using the program named **Graph06** to plot the output for the program named **Dsp034** using default parameters.

*(The file named **Dsp034.txt** did not exist in the current directory when the results shown in [Figure 2](#) were generated.)*

Figure 2. Spectral analysis of a damped pulse.



The input pulse

First consider the input pulse shown in the top plot of [Figure 2](#). This is the sort of pulse that you would get if you hung a mass on a spring, gave it a swift downward kick, and then allowed the mass to come to rest in an unimpeded fashion.

The horizontal axis represents time moving from left to right. The values above and below the axis represent the position of the mass over time relative to its position at rest.

Potential energy in the spring

When the mass is at the most extreme positions and getting ready to reverse directions, the spring is extended. Thus, potential energy is stored in the spring. At these points in time, there is no kinetic energy stored in the mass.

Kinetic energy in the mass

As the mass goes through the rest position heading towards the other side, the spring is no longer extended, and there is no potential energy stored in the spring. However, at that point in time, the mass is moving causing it to have kinetic energy.

An exchange of energy

The behavior of the spring/mass system is to exchange that energy between potential energy and kinetic energy until such time as energy losses due to friction dissipate the energy. At that point in time, the entire system comes to rest, which is the case about one-third of the way across the top plot in [Figure 2](#).

Now that our physics lesson for the day is over, let's get back to programming and digital signal processing.

Capturing the position of the mass as a sampled time series

Assume that you use an analog to digital converter to capture the position of the mass at a set of uniformly spaced intervals in time and then you plot those samples along a time axis. You should see something resembling the top plot in [Figure 2](#).

Having captured that positional information as a set of uniformly spaced samples, you are now in a position to perform a Fourier transform on the sampled time series.

Computing the Fourier transform

In the module titled [Fun with Java, How and Why Spectral Analysis Works](#), I explained that you could compute the Fourier transform of an input time series at any given frequency F by evaluating the first two expressions in [Figure 3](#). (The notation used in [Figure 3](#) was also explained in that module.)

Figure 3. Fourier transform equations.

$$\text{Real}(F) = S(n=0, N-1)[x(n) \cdot \cos(2\pi F \cdot n)]$$

$$\text{Imag}(F) = S(n=0, N-1)[x(n) \cdot \sin(2\pi F \cdot n)]$$

$$\text{ComplexAmplitude}(F) = \text{Real}(F) - j \cdot \text{Imag}(F)$$

I also explained that the Fourier transform of the input time series at that frequency can be viewed as a complex number having a *real part* and an *imaginary part* as in the third expression in [Figure 3](#).

(The amplitude of the spectrum at that frequency can be determined by computing the square root of the sum of the squares of the real and imaginary parts at that frequency but that isn't of major interest in this module.)

The Fourier transform of an input time series can be computed by performing these calculations across a range of frequencies.

The results of performing a spectral analysis on the pulse

The bottom four plots in [Figure 2](#) show the results of performing a Fourier transform on the pulse in the top plot.

(In this display format, which was produced by the program named Graph06, each sample value is represented by a vertical bar whose height is proportional to the value of the sample.)

These four plots show the values of the Fourier transform output at a set of uniformly spaced frequencies ranging from zero to 0.25 times the sampling frequency.

The amplitude spectrum

The second plot from the top in [Figure 2](#) shows the value of the amplitude spectrum. This is the Fourier transform output that we have been using in the previous modules in this series.

(Those modules ignored the complex spectrum and the phase angle.)

As you can see, the amplitude spectrum peaks at a frequency equal to 0.0625 times the sampling frequency. The reason for this will become clear when we examine the code that produced the pulse shown in the first plot.

The real and imaginary parts of the transform

The real part of the transform is shown in the third plot and the imaginary part of the transform is shown in the fourth plot. *(I believe that this is the first time that I have presented the real and imaginary parts of the spectrum in this series of modules.)*

The phase angle in degrees

The phase angle in degrees is shown in the bottom plot. There are a variety of different ways to display phase angles. This program displays the phase angle as values that range from -180 degrees to +180 degrees.

(It is also possible to display the phase angle as ranging from 0 to 360 degrees, or any combination that equates to 360 degrees or one full rotation. It is also possible to display the phase angle in radians instead of degrees.)

How is the phase angle computed?

Basically, the phase angle is the angle that you get when you compute the arc tangent of the ratio of the imaginary part to the real part of the complex spectrum at a particular frequency. However, beyond computing the arc tangent, you must do some additional work to take the quadrant into account.

How should we interpret the phase angle?

To begin with, you should ignore the result of phase angle computations at those frequencies for which there is insignificant energy. It is always possible to form a ratio of the values of the real and imaginary parts of the complex Fourier transform at any frequency. However, if the real and imaginary values produced by the Fourier transform at that frequency are both very small, the phase angle resulting from that ratio is of no practical significance. In fact, the angle can be corrupted by arithmetic errors resulting from performing arithmetic on very small values.

Therefore, noting the amplitude spectrum in the second plot of [Figure 2](#), phase angle results to the right of the fourth tick mark are probably useless.

Many combinations

The phase angle produced by performing a Fourier transform on a pulse of a given waveform is not unique. There are an infinite number of combinations of real and imaginary parts that can result from performing a Fourier transform on a given waveform, depending on how you define the origin of time. This means that there are also an infinite number of phase angle curves that can be produced from the ratio of those real and imaginary parts. I will explain this in more detail later using simpler pulses.

The frequency band of primary interest

In the case shown in [Figure 2](#), the frequency band of primary interest lies approximately between the first and the third tick marks. Most of the energy can be seen to lie between those limits on the basis of the amplitude plot.

The phase angle curve goes from a little more than zero degrees to a little less than 180 degrees across this frequency interval. However, it is significant to note that the phase angle is not linear across this frequency interval. Rather the shape of the curve is more like an elongated S sloping to the right.

A nonlinear phase angle, so what?

What is the significance of the nonlinear phase angle? If this plot represented the frequency response of your audio system, the existence of the nonlinear phase angle would be bad news. In particular, it would mean that the system would introduce phase distortion into your favorite music.

Computation of the phase angle

In an earlier module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#), I explained most of the code in the method named **transform** belonging to the class named **ForwardRealToComplex01**. However, I skipped over that portion of the code that computes the phase angle on the basis of the values of the real and imaginary parts. I am going to explain that code in this module. For an explanation of the rest of the code in the **transform** method, go back and

review the module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#).

The code in the **transform** method that computes the phase angle for a particular frequency is shown in [Listing 1](#). At this point in the execution of the **transform** method, the values of the real part (*real*) and the imaginary part (*imag*) of the Fourier transform at a particular frequency have been computed. Those values are used to compute the phase angle at that frequency.

Listing 1. Computation of the phase angle.

```
        if(imag == 0.0 && real == 0.0){ang = 0.0;}
        else{ang = Math.atan(imag/real)*180.0/pi;}

        if(real < 0.0 && imag == 0.0){ang =
180.0;}
        else if(real < 0.0 && imag == -0.0){
                                ang =
-180.0;}
        else if(real < 0.0 && imag > 0.0){
                                ang +=
180.0;}
        else if(real < 0.0 && imag < 0.0){
                                ang +=
-180.0;}
        angleOut[i] = ang;
```

What if both values are zero?

The code begins by testing to see if both the real and imaginary parts are equal to zero. If so, attempting to form the ratio of the imaginary part to the real part would be meaningless. In this case, the code in [Listing 1](#) simply sets the phase angle to a value of zero.

The `atan` method of the `Math` class

If both the real and imaginary parts are not zero, then the ratio of the imaginary value to the real value is formed and passed as a parameter to the **`atan`** method of the **`Math`** class.

The **`atan`** method returns the angle in radians whose tangent matches the value received as a parameter. The angle that is returned is in the range from $-\pi/2$ to $+\pi/2$ (*-90 degrees to +90 degrees*). The code in [Listing 1](#) multiplies that angle in radians by $180.0/\pi$ to convert the angle from radians to degrees.

Correction required for the quadrant

Although we have an intermediate answer at this point, we're still not finished. There is more work to do. The **`atan`** method simply uses the sign of its incoming parameter to decide whether to report the angle as positive or negative, and it only covers angles in two quadrants (*-90 degrees to +90 degrees*). We know that the angle can actually be in any one of four quadrants (*-180 degrees to +180 degrees*).

The first and third quadrants

For example, a positive ratio can result from a positive imaginary value and a positive real value, or from a negative imaginary value and a negative real value. Both of these would be reported by the **`atan`** method as being between 0 and 90 degrees when in fact, the negative imaginary value and the negative real value means that the angle is actually between -90 degrees and -180 degrees.

The second and fourth quadrants

Similarly, a negative ratio can result from a negative imaginary value and a positive real value or from a positive imaginary value and a negative real value. Both of these would be reported by the **atan** method as being between 0 and -90 degrees when in fact, the positive imaginary value and the negative real value means that the angle is actually between 90 degrees and 180 degrees.

An exercise for the reader

I will leave it as an exercise for the reader to work through the remaining code in [Listing 1](#) to see how this code determines the proper quadrant and adjusts the angle appropriately, all the while maintaining the angle between -180 degrees and +180 degrees.

The program named Dsp034

I will discuss the program named **Dsp034** in fragments. A complete listing of the program is provided in [Listing 6](#) near the end of the module.

Due to the similarity of this program to programs explained in previous modules in this series, this discussion will be rather brief. Following a discussion of the code, I will provide and explain some more spectral analysis results obtained by running the program with parameters read from the file named **Dsp034.txt** .

The beginning of the Dsp034 class

The beginning of the class, along with the declaration of several variables is shown in [Listing 2](#).

(Note that the class implements the interface named GraphIntfc01.)

The variable names and the embedded comments should make the purpose of these variables self explanatory. If not, you will see how they are used later in the program.

Listing 2. The beginning of the Dsp034 class.

```
class Dsp034 implements GraphIntfc01{
    final double pi = Math.PI;//for simplification

    int dataLen;//data length
    int pulseLen;//length of the pulse
    int zeroTime;//sample that represents zero
time
    //Low and high frequency limits for the
    // spectral analysis.
    double lowF;
    double highF;
    double[] pulse;//data describing the pulse

    //Following array stores input data for the
    // spectral analysis process.
    double[] data;

    //Following arrays receive information back
    // from the spectral analysis process
    double[] real;
    double[] imag;
    double[] angle;
    double[] mag;
```

The constructor

The constructor begins in [Listing 3](#). The code in the constructor either calls the `getParameters` method to get the operating parameters from the file named `Dsp034.txt`, or creates a set of operating parameters on the fly if that file does not exist in the current directory. In either case, many of the variables declared in [Listing 2](#) are populated as a result of that action.

Listing 3. Beginning of the constructor.

Listing 3. Beginning of the constructor.

```
public Dsp034(){//constructor

    //Get the parameters from a file named
    // Dsp034.txt. Create and use default
    // parameters describing a damped sinusoidal
    // pulse if the file doesn't exist in the
    // current directory.
    if(new File("Dsp034.txt").exists()){
        getParameters();
    }else{
        //Create default parameters
        dataLen = 400;//data length
        pulseLen = 100;//pulse length
        //Sample that represents zero time.
        zeroTime = 0;
        //Low and high frequency limits for the
        // spectral analysis.
        lowF = 0.0;
        highF = 0.5;//half the sampling frequency
        pulse = new double[pulseLen];
        double scale = 240.0;
        for(int cnt = 0;cnt < pulseLen;cnt++){
            scale = 0.94*scale;//damping scale
factor
            pulse[cnt] =

scale*Math.sin(2*pi*cnt*0.0625);
        }//end for loop
        //End default parameters
    }//end else
```

For the case where the file named **Dsp034.txt** doesn't exist in the current directory, the code in the **else** clause in [Listing 3](#) establishes default operating parameters, and creates the damped sinusoid pulse shown in the top plot of [Figure 2](#).

Store the pulse in the time series

At this point in the process, the array referred to by the reference variable named **pulse** contains a set of samples that constitutes a pulse. The code in [Listing 4](#) creates a data array containing the data upon which spectral analysis will be performed and stores the pulse in that array.

(All elements in the data array other than those elements occupied by values of the pulse have a value of zero.)

Listing 4. Store the pulse in the time series.

```
data = new double[dataLen];  
for(int cnt = 0;cnt < pulse.length;cnt++){  
    data[cnt] = pulse[cnt];  
}//end for loop
```

Print the parameters and the pulse

Following this, code in the constructor prints the parameters and the values of the samples that constitute the pulse. You can view that code in [Listing 6](#) near

the end of the module.

Perform the spectral analysis

Finally, the code in [Listing 5](#) creates the array objects that will receive the results of the spectral analysis, and calls the **transform** method of the **ForwardRealToComplex01** class to perform the spectral analysis.

Listing 5. Perform the spectral analysis.

```
mag = new double[dataLen];
real = new double[dataLen];
imag = new double[dataLen];
angle = new double[dataLen];
ForwardRealToComplex01.transform(data, real,
    imag, angle, mag, zeroTime, lowF, highF);

} //end constructor
```

[Listing 5](#) also signals the end of the constructor. At this point, the object has been instantiated and its array objects have been populated with the input and output data from the spectral analysis process. This data is ready to be handed over to the plotting program to be plotted, as shown in [Figure 2](#).

The getParameters method

The **getParameters** method, called in [Listing 3](#), reads parameter and pulse data from the file named **Dsp034.txt**, and deposits that data into the variables and the pulse array declared in [Listing 2](#).

The code in the **getParameters** method is straightforward, so I won't bore you with an explanation. You can view the method in [Listing 6](#) near the end of the module.

The interface methods

As pointed out earlier, the **Dsp034** class implements the **GraphIntfc01** interface. As such, the class must define the six methods declared in that interface. These methods are called by the plotting program to obtain the data that is to be plotted.

You have seen implementations of these methods in several earlier modules, so there is nothing new here. Consequently, I won't discuss the interface methods. You can view the methods in [Listing 6](#) near the end of the module.

The one thing that you might want to pay attention to in these methods is the scaling that is applied to the data before it is returned. This is an attempt to cause all of the curves to plot reasonably well within a value range of -180 to +180. This range is dictated by the fact that this is the range of values for the phase angle data.

Now that you understand the inner workings of the program, let's look at some more examples, this time getting the input data from the file named **Dsp034.txt** .

More examples

The simplest pulse of all, an impulse

The simplest pulse that you can create is a single non-zero valued sample among a bunch of zero-valued samples. This simple pulse, commonly called an impulse in digital signal processing, is an extremely important type of signal. It is used for a variety of purposes in testing both digital and analog signal processing systems.

Let's examine the result of performing spectral analysis on an impulse.

The parameters used for this experiment are shown in [Figure 4](#).

Figure 4. The simplest pulse of all, an impulse.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
180.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```

Screen output from the `getParameters` method

The data that you see in [Figure 4](#) is the screen output from the method named **getParameters**. To orient you with this format, the second item in [Figure 4](#) indicates that the entire length of the data upon which spectral analysis was

performed was 400 samples. All 400 samples had a value of zero except for the values of the sample in the pulse that occurred at the beginning. In this program, setting the total data length to 400 causes the spectral analysis to be performed at 400 individual frequencies.

(By now, you may be suspecting that I have a particular affinity for a data length of 400 samples. If so, you are correct. This is not a technical affinity. Rather, it is a visual one. These figures are formatted such that the plotted data occupies an area of the screen containing approximately 400 pixels. By matching the plotted points to the positions of the pixels, it is possible to avoid, or at least minimize, the distortion that can occur when attempting to map from sample points to pixel locations when there is a mismatch between the two.)

To see the result of such mapping problems, repeat the experiment shown in [Figure 2](#) and use your mouse to stretch the Frame horizontally by a very small amount. Depending on how much you stretch the Frame, you should see vertical lines disappear, vertical lines that are too close together, or a combination of the two. This is another manifestation of the impact of sampling that I don't have the time to get into at this point.)

The length of the pulse

As you can see in [Figure 4](#), the length of the pulse for this experiment was 11 samples, all but one of which had a value of zero.

(In this case, I could have made the pulse length 1 but for simplicity, I will keep it at 11 for several different experiments.)

Defining the origin of time

As you may have discovered by playing video games, we can do things with a computer that we can't do in the real world. For example, the Fourier transform program allows me to specify which sample I regard as representing zero time. Samples to the left of that sample represent negative time (*history*) and samples to the right of that one represent positive time (*the future*) .

In this case, I specified that the first sample (*sample number 0*) represents zero time. As you will see later, this has a significant impact on the distribution of energy between the real and imaginary parts of the transform results, and as such, has a significant impact on the phase angle.

Computational frequency range

Because I am using a DFT algorithm (*instead of an FFT algorithm*) I can compute the Fourier transform across a range of frequencies of my choosing. As shown in [Figure 4](#), I chose to compute the transform across the range of frequencies from zero to one-half the sampling frequency, known as the Nyquist folding frequency. Thus, the spectral analysis was performed at 400 individual frequencies between these two limits.

The values that make up the pulse

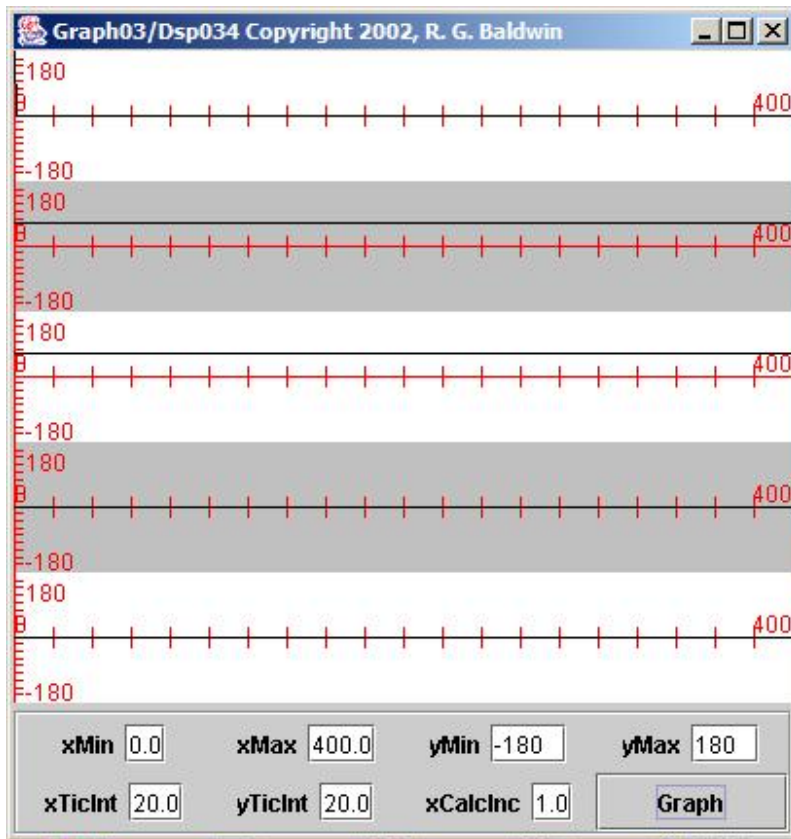
The computational frequency range is followed in [Figure 4](#) by the values of the samples that make up the pulse. Note that the first sample has a value of 180 while the other ten samples all have a value of 0.

Results of spectral analysis on an impulse

I used the program named **Graph03** to plot the output from this program, and the results are shown in [Figure 5](#).

(Note that rather than plotting each sample value as a vertical bar, **Graph03** plots each sample as a dot and connects the dots with straight line segments.)

Figure 5. Spectral analysis of an impulse at zero time.



Where is that impulse?

If you strain your eyes and you know exactly where to look, you may be able to see a single black spike with an amplitude of 180 at the far left of the top plot. If you can't see it, you will simply have to trust me when I tell you that it is there.

The amplitude spectrum

Now take a look at the amplitude spectrum in the second plot from the top. What you should see is a straight black line extending from zero to the folding frequency on the right. This is because such an impulse (*theoretically*) contains an equal distribution of energy at every frequency from zero to infinity.

(In reality, there is no such thing as a perfect impulse, so there is no such thing as infinite bandwidth. However, the bandwidth of a practical impulse is very wide and the amplitude spectrum is very flat.)

That is one of the things that make the impulse so useful. It is often used for various testing purposes in both the analog world and the digital world.

The real spectrum

Now look at the real spectrum in the second plot from the top. As you can see, it looks exactly like the amplitude spectrum. This is because the impulse appears at zero time. We will change this in the next experiment so that you can see the impact of a time delay on the complex spectrum.

The imaginary spectrum

Moving on down the page, the imaginary part of the spectrum is a flat line with a value of zero across the entire frequency range. Once again, this is

because the impulse appears at zero time.

The phase angle

Because the imaginary value is zero everywhere, the ratio of the imaginary value to the real value is also zero everywhere. Thus, the phase angle is also zero at all frequencies within the range.

Introduce a time delay

Now we are going to introduce a one-sample time delay in the location of the impulse relative to the time origin. We will keep the zero time reference at the first sample and cause the impulse to appear as the second sample in the eleven-sample sequence.

The new parameters are shown in [Figure 6](#). The only change is the move of the impulse from the first sample in the eleven-sample pulse to the second sample in the eleven-sample pulse.

Figure 6. Introduce a time delay.

Figure 6. Introduce a time delay.

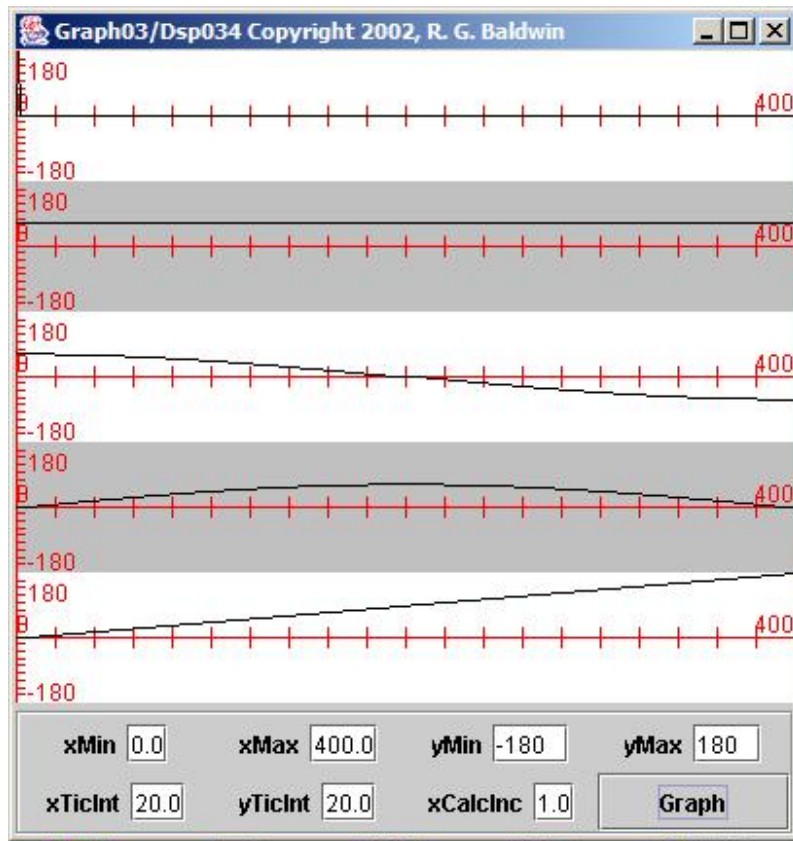
```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
0.0
180.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```

The spectral analysis output

The result of performing the spectral analysis on this new time series is shown in [Figure 7](#).

Figure 7. Spectral analysis of an impulse with a one-sample delay.

Figure 7. Spectral analysis of an impulse with a one-sample delay.



Once again, if you strain your eyes, you may be able to see the impulse on the left end of the top plot. It has been shifted one sample to the right relative to that shown in [Figure 5](#).

The amplitude spectrum

The amplitude spectrum in the second plot looks exactly like it looked in [Figure 5](#). That is as it should be. The spectral content of a pulse is determined by its waveform, not by its location in time. Simply moving the impulse by one sample into the future doesn't change its spectral content.

The real and imaginary parts of the spectrum

However, moving the impulse one sample into the future (*a time delay*) did change the values of the real and imaginary parts of the complex spectrum. As you can see from the third plot in [Figure 7](#), the real part of the spectrum is no longer a replica of the amplitude spectrum. Also the imaginary part of the spectrum in the fourth plot is no longer zero across the frequency range from zero to the folding frequency.

Real and imaginary values are intrinsically linked

However, the real and imaginary parts cannot change in arbitrary ways relative to one another. Recall that the amplitude spectrum at each individual frequency is the square root of the sum of the squares of the real and imaginary parts. In order for the amplitude spectrum to stay the same, changes to the real part of the spectrum must be accompanied by changes to the imaginary part that will maintain that relationship.

The phase angle spectrum

Finally, the phase angle shown in the bottom plot is no longer zero. Rather it is a straight line with a value of zero at zero frequency and a value of 180 degrees at the folding frequency.

An important conclusion

Simply shifting an impulse forward or backward in time introduces a phase shift that is linear with frequency. Shifting the pulse forward in time introduces a linear phase shift with a positive slope. Shifting the pulse backwards in time introduces a linear phase shift with a negative slope. In both cases, the amount of slope depends on the amount of time shift.

The converse is also true

A shift in time introduces a linear phase shift. Conversely, introducing a linear phase shift causes a shift in time.

A more acceptable form of phase distortion

Once again, consider your audio system. If your audio system introduces a phase shift across the frequency band of interest, you would probably like for that phase shift to be linear with frequency. That will simply cause the music to be delayed in time. In other words, all frequency components in the music will be delayed an equal amount of time.

On the other hand, if the phase shift is not linear with frequency, some frequencies will be delayed more than other frequencies. This sometimes results in noticeable phase distortion in your music.

Introduce a large time delay

Now let's move the impulse to the center of the eleven-sample pulse and observe the result. The new parameters are shown in [Figure 8](#).

Figure 8. Introduce a large time delay.

Figure 8. Introduce a large time delay.

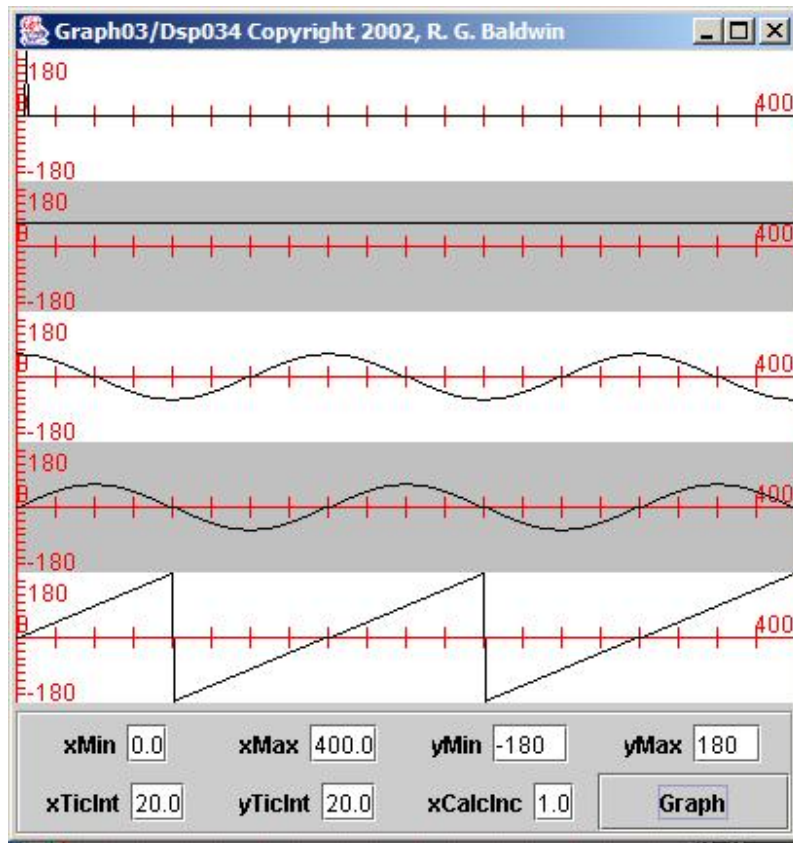
```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
0.0
0.0
0.0
0.0
0.0
0.0
180.0
0.0
0.0
0.0
0.0
0.0
```

The output from the spectral analysis

[Figure 9](#) shows the result of performing a spectral analysis on the time series containing this new time-delayed impulse.

Figure 9. Spectral analysis of impulse with five-sample delay.

Figure 9. Spectral analysis of impulse with five-sample delay.



Finally, you can probably see the impulse on the left side of the top plot in [Figure 9](#) without straining your eyes too much.

As we would expect, the amplitude spectrum hasn't changed.

Although it wasn't apparent in [Figure 7](#), [Figure 9](#) shows that the real part of the spectrum takes on the shape of a cosine wave, while the imaginary part of the spectrum takes on the shape of a sine wave as a result of the time delay of the impulse.

The phase shift is still linear across frequency as would be expected, but the slope is now five times greater than the slope of the phase shift in [Figure 7](#).

(Note that the time delay is five times greater in [Figure 9](#). Note also that the plot of the phase angle wraps around from +180 degrees to -180 degrees each time the phase angle reaches +180 degrees. This produces the saw tooth effect shown in the bottom plot in [Figure 9](#).)

A boxcar pulse

If an impulse is the simplest kind of pulse to generate digitally, a boxcar pulse is probably the next simplest. A "boxcar" pulse is one where several adjacent samples have the same non-zero value. Let's examine the case for an eleven-sample boxcar pulse. The new parameters for this case are shown in [Figure 10](#).

Figure 10. A boxcar pulse.

Figure 10. A boxcar pulse.

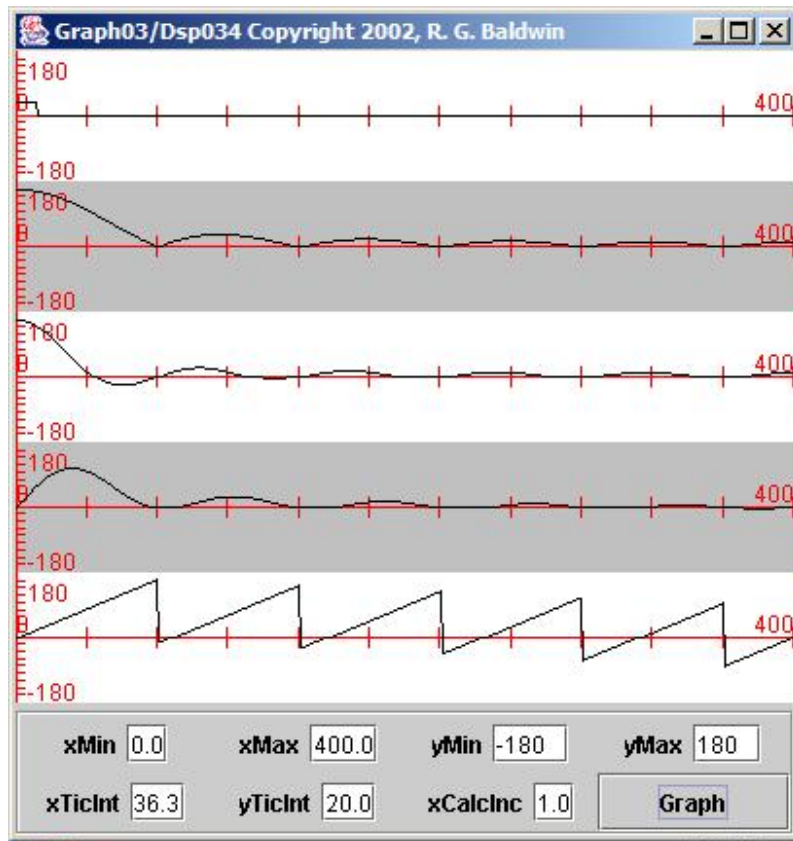
```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 0
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
40.0
```

The spectral analysis output for the boxcar pulse

The spectral analysis output for the eleven-sample boxcar pulse is shown in [Figure 11](#).

Figure 11. Spectral analysis of 11-sample boxcar pulse.

Figure 11. Spectral analysis of 11-sample boxcar pulse.



The pulse itself is relatively easy to see on the leftmost end of the top plot.

The amplitude spectrum

The amplitude spectrum is no longer flat. Rather it has a peak at zero frequency and goes to zero between frequency sample 72 and frequency sample 73.

Without getting into the technical details, I will simply tell you that the location of the point where it goes to zero is related to the reciprocal of the pulse width. If that sounds familiar, it is because we encountered similar situations involving bandwidth in the module titled [Spectrum Analysis using Java, Frequency Resolution versus Data Length](#).

In fact, the shape of the amplitude spectrum is a familiar $(\sin x)/x$ curve with the negative lobes flipped up and turned into positive lobes instead.

The phase angle is still linear with frequency although it now shows some discontinuities at those frequencies where the amplitude spectrum touches zero.

The magic of non real-time digital processing

When working with (*recorded non real-time*) digital time series, it is not only possible to physically shift pulses forward or backward in time, it is also possible to leave the pulses where they are and redefine the underlying time base. For the next experiment, I will leave everything else the same and redefine the location of the origin of time. I will place the time origin at the middle of the boxcar pulse.

The new parameters for this experiment are shown in [Figure 12](#). Note that the only significant difference between [Figure 12](#) and [Figure 10](#) is the redefinition of the sample that represents zero time. I redefined the time origin from sample 0 to sample 5. This causes the boxcar pulse to be centered on zero time. Five of the samples in the boxcar pulse occur in negative (*history*) time. One sample occurs exactly at zero time. The other five samples occur in positive (*future*) time .

Figure 12. Shift the time base.

Figure 12. Shift the time base.

```
Parameters read from file
Data length: 400
Pulse length: 11
Sample for zero time: 5
Lower frequency bound: 0.0
Upper frequency bound: 0.5
Pulse Values
40.0
40.0
40.0
40.0
40.001
40.0
40.0
40.0
40.0
40.0
40.0
```

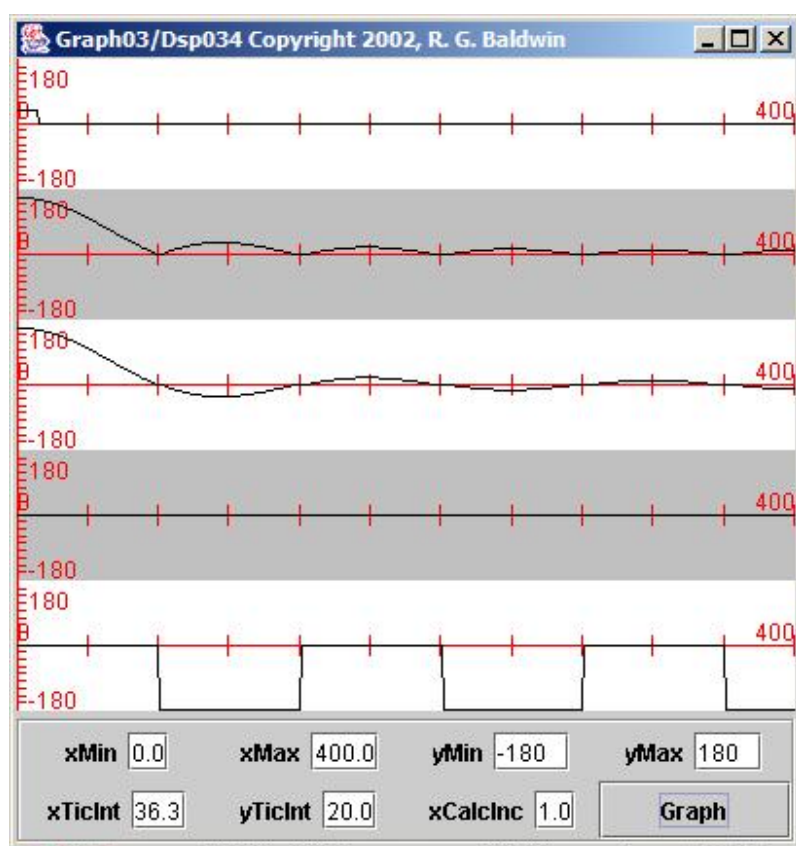
(I did make one other change. This change was to add a tiny spike to one of the samples near the center of the pulse. This creates a tiny amount of wide-band energy and tends to stabilize the computation of the phase angle. It prevents the imaginary part of the spectrum from switching back and forth between very small positive and negative values due to arithmetic errors.)

The spectral analysis output

The output from the spectral analysis is shown in [Figure 13](#). The magnitude spectrum hasn't changed. The real part of the spectrum has changed

significantly. It is now a true $(\sin x)/x$ curve with both positive and negative lobes.

Figure 13. Spectral analysis of 11-sample boxcar centered in time.



The imaginary part of the spectrum is zero or nearly zero at every frequency. *(It would be zero in the absence of arithmetic errors.)*

The phase angle is zero across the entire main energy lobe of the spectrum. It is -180 degrees in those frequency areas where the real part of the spectrum is negative, and is zero in those frequency areas where the real part of the

spectrum is positive. There is no linear phase shift because the boxcar pulse is centered on the time origin.

Probably more than you ever wanted to know

And that is probably more than you ever wanted to know about the complex spectrum, phase angles, and time shifts. I will stop writing and leave it at that.

Run the program

I encourage you to copy, compile, and run the program provided in this module. Experiment with it, making changes and observing the results of your changes.

Create more complex experiments. For example, you could create pulses of different lengths with complex shapes and examine the complex spectra and phase angles for those pulses.

If you really want to get fancy, you could create a pulse consisting of a sinusoid whose frequency changes with time from the beginning to the end of the pulse. (*A pulse of this type is often referred to as a frequency modulated sweep signal.*) See what you can conclude from doing spectral analysis on a pulse of this type. Pay particular attention to the phase angle across the frequency band containing most of the energy.

Most of all enjoy yourself and learn something in the process.

Summary

The default pulse for the **Dsp034** program is a damped sinusoid. This is a pulse whose shape is commonly found in mechanical and electronic systems in the real world. The phase angle in the complex spectrum for a pulse of this shape is nonlinear. Among other things, nonlinear phase angles introduce phase distortion into audio systems.

The simplest pulse of all is a single impulse. A pulse of this type has an infinite bandwidth (*theoretically*) and a linear phase angle. The slope of the

phase angle depends on the location of the pulse relative to the time origin.

Shifting a pulse in time introduces a linear phase angle to the complex spectrum. Conversely, introducing a linear phase angle to the complex spectrum causes a pulse to be shifted in time.

What's next?

The next module in this series will introduce the *inverse* Fourier transform (*as opposed to the forward Fourier transform*) and will explain the reversible nature of the Fourier transform.

Complete program listing

A complete listing of the program discussed in this module is provided in Listing 6 below. Listings for other programs mentioned in the module, such as **Graph03** and **Graph06**, are provided in other modules. Those modules are identified in the text of this module.

Listing 6. Dsp034.java.

```
/* File Dsp034.java  
Copyright 2004, R.G.Baldwin  
Rev 5/21/04
```

```
Computes and plots the amplitude, real, imag,  
and phase angle spectrum of a pulse that is read  
from a file named Dsp034.txt. If that file  
doesn't exist in the current directory, the  
program uses a set of default parameters  
describing a damped sinusoidal pulse.
```

Listing 6. Dsp034.java.

The program also plots the pulse. When the data is plotted using the programs Graph03 or Graph06, the order of the plots from top to bottom in the display are:

The pulse

The amplitude spectrum

The real spectrum

The imaginary spectrum

The phase angle in degrees

Each parameter value must be stored as characters on a separate line in the file named Dsp034.txt.

The required parameters are as follows:

Data length as type int

Pulse length as type int

Sample number representing zero time as type int

Low frequency bound as type double

High frequency bound as type double

Sample values for the pulse as type double

The number of sample values must match the value for the pulse length.

All frequency values are specified as a double representing a fractional part of the sampling frequency. For example, a value of 0.5 specifies a frequency that is half the sampling frequency.

Here is a set of sample parameter values that can be used to test the program. This sample data describes a triangular pulse. Don't allow blank lines at the end of the data in the file.

Listing 6. Dsp034.java.

```
11
0
0.0
0.5
0
0
0
45
90
135
90
45
0
0
0
```

The plotting program that is used to plot the output data from this program requires that the program implement `GraphIntfc01`. For example, the plotting program named `Graph03` can be used to plot the data produced by this program. When it is used, the usage information is:

```
java Graph03 Dsp034
```

A static method named `transform` belonging to the class named `ForwardRealToComplex01` is used to perform the actual spectral analysis. The method named `transform` does not implement an FFT algorithm. Rather, it is more general than, but much slower than an FFT algorithm. (See the program named `Dsp030` for the use of an FFT algorithm.)

Tested using SDK 1.4.2 under WinXP.

Listing 6. Dsp034.java.

```
*****/  
import java.util.*;  
import java.io.*;  
  
class Dsp034 implements GraphIntf01{  
    final double pi = Math.PI;//for simplification  
  
    int dataLen;//data length  
    int pulseLen;//length of the pulse  
    int zeroTime;//sample that represents zero time  
    //Low and high frequency limits for the  
    // spectral analysis.  
    double lowF;  
    double highF;  
    double[] pulse;//data describing the pulse  
  
    //Following array stores input data for the  
    // spectral analysis process.  
    double[] data;  
  
    //Following arrays receive information back  
    // from the spectral analysis process  
    double[] real;  
    double[] imag;  
    double[] angle;  
    double[] mag;  
  
    public Dsp034(){//constructor  
  
        //Get the parameters from a file named  
        // Dsp034.txt. Create and use default  
        // parameters describing a damped sinusoidal  
        // pulse if the file doesn't exist in the  
        // current directory.  
        if(new File("Dsp034.txt").exists()){  
            getParameters();  
        }  
    }  
}
```

Listing 6. Dsp034.java.

```
}else{
    //Create default parameters
    dataLen = 400;//data length
    pulseLen = 100;//pulse length
    //Sample that represents zero time.
    zeroTime = 0;
    //Low and high frequency limits for the
    // spectral analysis.
    lowF = 0.0;
    highF = 0.5;//half the sampling frequency
    pulse = new double[pulseLen];
    double scale = 240.0;
    for(int cnt = 0;cnt < pulseLen;cnt++){
        scale = 0.94*scale;//damping scale factor
        pulse[cnt] =
            scale*Math.sin(2*pi*cnt*0.0625);
    }//end for loop
    //End default parameters
}//end else

//Create the data array and deposit the pulse
// in it.
data = new double[dataLen];
for(int cnt = 0;cnt < pulse.length;cnt++){
    data[cnt] = pulse[cnt];
}//end for loop

//Print parameter values.
System.out.println(
    "Data length: " + dataLen);
System.out.println(
    "Pulse length: " + pulseLen);
System.out.println(
    "Sample for zero time: " + zeroTime);
System.out.println(
    "Lower frequency bound: " + lowF);
```

Listing 6. Dsp034.java.

```
        System.out.println(
            "Upper frequency bound: " + highF);
    System.out.println("Pulse Values");
    for(int cnt = 0;cnt < pulseLen;cnt++){
        System.out.println(pulse[cnt]);
    }//end for loop

    //Create array objects to receive the results
    // and perform the spectral analysis.
    mag = new double[dataLen];
    real = new double[dataLen];
    imag = new double[dataLen];
    angle = new double[dataLen];
    ForwardRealToComplex01.transform(data,real,
        imag,angle,mag,zeroTime,lowF,highF);

} //end constructor
//-----//

//This method gets processing parameters from
// a file named Dsp034.txt and stores those
// parameters in instance variables belonging
// to the object of type Dsp034.
void getParameters(){

    int cnt = 0;
    //Temporary holding area for strings. Allow
    // space for a few blank lines at the end
    // of the data in the file.
    String[] data = new String[20];
    try{
        //Open an input stream.
        BufferedReader inData =
            new BufferedReader(new FileReader(
                "Dsp034.txt"));
        //Read and save the strings from each of
```

Listing 6. Dsp034.java.

```
// the lines in the file. Be careful to
// avoid having blank lines at the end,
// which may cause an ArrayIndexOutOfBoundsException
// exception to be thrown.
while((data[cnt] =
        inData.readLine()) != null){
    cnt++;
} //end while
inData.close();
} catch(IOException e){}

//Move the parameter values from the
// temporary holding array into the instance
// variables, converting from characters to
// numeric values in the process.
cnt = 0;
dataLen =
    (int)Double.parseDouble(data[cnt++]);
pulseLen =
    (int)Double.parseDouble(data[cnt++]);
zeroTime =
    (int)Double.parseDouble(data[cnt++]);
lowF = Double.parseDouble(data[cnt++]);
highF = Double.parseDouble(data[cnt++]);

//Create a new array object for the pulse
// and populate it from the file data.
pulse = new double[pulseLen];
for(int pCnt = 0;pCnt < pulseLen;pCnt++){
    pulse[pCnt] = Double.parseDouble(
        data[cnt++]);
} //end for loop
System.out.println(
    "Parameters read from file");

} //end getParameters
```

Listing 6. Dsp034.java.

```
//-----//
//The following six methods are required by the
// interface named GraphIntfc01. The plotting
// program pulls the data values to be plotted
// by calling these methods.
public int getNmbr(){
    //Return number of functions to process.
    // Must not exceed 5.
    return 5;
} //end getNmbr
//-----//
//Provide the input data for plotting.
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > data.length-1){
        return 0;
    }else{
        return data[index];
    } //end else
} //end function
//-----//
//Provide the amplitude spectral data for
// plotting. Attempt to scale it so that it
// will plot well in the range 0 to 180.
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > mag.length-1){
        return 0;
    }else{
        return (4*dataLen/pulseLen)*mag[index];
    } //end else
} //end function
//-----//
//Provide the real spectral data for
// plotting. Attempt to scale it so that it
// will plot well in the range -180 to 180.
```

Listing 6. Dsp034.java.

```
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > real.length-1){
        return 0;
    }else{
        //Scale for convenient display
        return (4*dataLen/pulseLen)*real[index];
    }//end else
}//end function
//-----//
//Provide the imaginary spectral data for
// plotting. Attempt to scale it so that it
// will plot well in the range -180 to 180.
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imag.length-1){
        return 0;
    }else{
        //Scale for convenient display
        return (4*dataLen/pulseLen)*imag[index];
    }//end else
}//end function
//-----//
//Provide the phase angle data for plotting.
// The angle ranges from -180 degrees to +180
// degrees. This is thing that drives the
// attempt to cause the other curves to plot
// well in the range -180 to +180.
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > angle.length-1){
        return 0;
    }else{
        return angle[index];
    }//end else
}//end function
```

Listing 6. Dsp034.java.

```
//-----//  
  
} //end class Dsp034
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1484-Spectrum Analysis using Java, Complex Spectrum and Phase Angle
- File: Java1484.htm
- Published: 09/21/04

Baldwin discusses the complex spectrum and explains the relationship between the phase angle and shifts in the time domain.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book,

please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1485-Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain

Baldwin illustrates and explains forward and inverse Fourier transforms using both DFT and FFT algorithms. He also illustrates and explains the implementation of frequency filtering by modifying the complex spectrum in the frequency domain and transforming the modified complex spectrum back into the time domain.

Revised: Sat Oct 17 17:00:21 CDT 2015

This page is included in the following book: [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Description of the program named Dsp035](#)
 - [The order of the plotted results](#)
 - [The format of the plots](#)
 - [The forward Fourier transform](#)
 - [The inverse Fourier transform](#)
 - [Results](#)
 - [The program named Dsp035](#)
 - [The beginning of the class named Dsp035](#)

- [Beginning of the constructor](#)
- [Compute the complex spectrum](#)
- [Perform the inverse Fourier transform](#)
- [The InverseComplexToReal01 class](#)
 - [Parameters for the inverseTransform method](#)
 - [Beginning of the inverseTransform method](#)
- [The program named Dsp036](#)
 - [The output from Dsp036](#)
 - [Some code from Dsp036](#)
 - [The signature of the complexToComplex method](#)
- [Using a Fourier transform to perform frequency filtering](#)
 - [Operation of the program](#)
 - [Beginning of the class for Dsp037](#)
 - [Beginning of the constructor](#)
 - [Compute the Fourier transform](#)
 - [Apply the filter to the frequency data](#)
 - [Re-compute the magnitude](#)
 - [Compute the inverse Fourier transform](#)
 - [Display the results](#)
- [One more example, Dsp038](#)
 - [Compare the results](#)
- [Run the programs](#)
- [Summary](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

A previous module titled [Fun with Java, How and Why Spectral Analysis Works](#) explained some of the fundamentals regarding spectral analysis.

The module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#) presented and explained several Java programs for doing spectral analysis, including both DFT programs and FFT programs. That module illustrated the fundamental aspects of spectral analysis that center around the sampling frequency and the Nyquist folding frequency.

The module titled [Spectrum Analysis using Java, Frequency Resolution versus Data Length](#) used similar Java programs to explain spectral frequency resolution.

The module titled [Spectrum Analysis using Java, Complex Spectrum and Phase Angle](#) explained issues involving the complex spectrum, the phase angle, and shifts in the time domain.

This module will illustrate and explain *forward* and *inverse* Fourier transforms using both DFT and FFT algorithms. I will also illustrate and explain the implementation of frequency filtering by modifying the complex spectrum in the frequency domain and then transforming the modified complex spectra back into the time domain.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Forward and inverse transform of a time series using DFT algorithm.
- [Figure 2.](#) Forward and inverse transform of a time series using FFT algorithm.
- [Figure 3.](#) The signature of the complexToComplex method.
- [Figure 4.](#) Filtering in the frequency domain.
- [Figure 5.](#) Filtering in the frequency domain.

Listings

- [Listing 1.](#) The beginning of the class named Dsp035.
- [Listing 2.](#) Beginning of the constructor.
- [Listing 3.](#) Compute the complex spectrum.
- [Listing 4.](#) Perform the inverse Fourier transform.
- [Listing 5.](#) Beginning of the class named InverseComplexToReal01.
- [Listing 6.](#) The inverse transform computation.
- [Listing 7.](#) Some code from Dsp036.
- [Listing 8.](#) Beginning of the class for Dsp037.
- [Listing 9.](#) Beginning of the constructor.
- [Listing 10.](#) Compute the Fourier transform.
- [Listing 11.](#) Apply the filter to the frequency data.
- [Listing 12.](#) Re-compute the magnitude.
- [Listing 13.](#) Compute the inverse Fourier transform.
- [Listing 14.](#) Dsp035.java.
- [Listing 15.](#) InverseComplexToReal01.hava.
- [Listing 16.](#) Dsp036.java.
- [Listing 17.](#) InverseComplexToRealFFT01.java,
- [Listing 18.](#) Dsp037.java.
- [Listing 19.](#) Dsp038.java.

Preview

In this module, I will present and explain the following new programs:

- **Dsp035** - Illustrates the reversible nature of the Fourier transform. This program transforms a real time series into a complex spectrum, and then reproduces the real time series by performing an inverse Fourier transform on the complex spectrum. This is accomplished using a DFT algorithm.
- **InverseComplexToReal01** - Class that implements an inverse DFT algorithm for transforming a complex spectrum into a real time series.
- **Dsp036** - Replicates the behavior of the program named Dsp035 but uses an FFT algorithm instead of a DFT algorithm.
- **InverseComplexToRealFFT01** - Class that implements an inverse FFT algorithm for transforming a complex spectrum into a real time series.

- **Dsp037** - Illustrates filtering in the frequency domain. Uses an FFT algorithm to transform a time-domain impulse into the frequency domain. Modifies the complex spectrum, eliminating energy within a specific band of frequencies. Uses an inverse FFT algorithm to produce the filtered version of the impulse in the time domain.

In addition, I will use the following programs that I explained in the module titled [Spectrum Analysis using Java](#), [Sampling Frequency](#), [Folding Frequency](#), [and the FFT Algorithm](#) and other previous modules.

- **ForwardRealToComplex01** - Class that implements a forward DFT algorithm for transforming a real time series into a complex spectrum.
- **ForwardRealToComplexFFT01** - Class that implements a forward FFT algorithm for transforming a real time series into a complex spectrum.
- **Graph03** - Used to display various types of data. (The concepts were explained in an earlier module.)
- **Graph06** - Also used to display various types of data in a somewhat different format. (The concepts were also explained in an earlier module.)
- **GraphIntfc01** - An interface that is required by **Graph03** and **Graph06**

Discussion and sample code

Description of the program named Dsp035

The program named **Dsp035** illustrates *forward* and *inverse* Fourier transforms using DFT algorithms.

The program performs spectral analysis on a time series consisting of pulses and a sinusoid. Then it passes the resulting real and complex parts of the spectrum to an inverse Fourier transform program. This program performs an inverse Fourier transform on the complex spectral data to reconstruct the original time series.

This program can be run with either **Graph03** or **Graph06** in order to plot the results. Enter the following at the command-line prompt to run the program with **Graph03** after everything is compiled:

java Graph03 Dsp035

The program was tested using JDK 1.8 under Windows 7.

The order of the plotted results

When the data is plotted (see [Figure 1](#)) using the programs **Graph03** or **Graph06**, the plots appear in the following order from top to bottom:

- The input time series
- The real spectrum of the input time series
- The imaginary spectrum of the input time series
- The amplitude spectrum of the input time series
- The output time series produced by the inverse Fourier transform

The format of the plots

There were 256 values plotted horizontally in each section. I plotted the values on a grid that is 270 units wide to make it easier to view the plots on the rightmost end. This leaves some blank space on the rightmost end to contain the numbers, preventing the numbers from being mixed in with the plotted values. The last actual data value coincides with the rightmost tick mark on each plot.

The forward Fourier transform

A static method named **transform** belonging to the class named **ForwardRealToComplex01** was used to perform the forward Fourier transform.

*(I explained this class and the **transform** method in the earlier module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#).)*

The method named **transform** does not implement an FFT algorithm. Rather, it implements a DFT algorithm, which is more general than, but much slower than an FFT algorithm.

*(See the program named **Dsp036** later in the module for the use of an FFT algorithm.)*

The inverse Fourier transform

A static method named **inverseTransform** belonging to the class named **InverseComplexToReal01** was used to perform the inverse Fourier transform. I will explain this method later in this module.

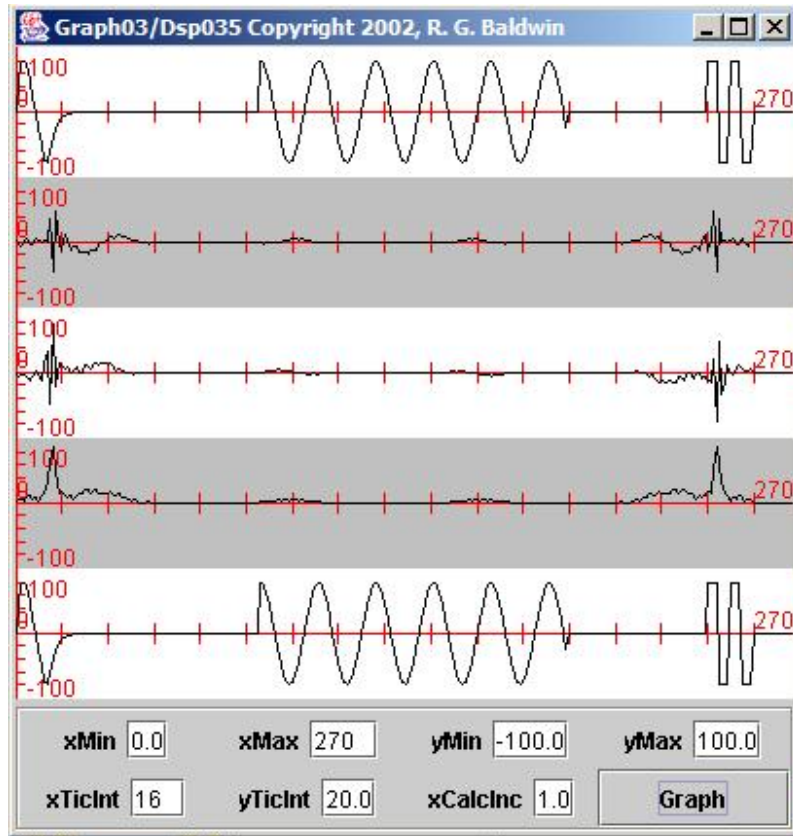
Results

Before getting into the technical details of the program, let's take a look at the results shown in [Figure 1](#).

The top plot in [Figure 1](#) shows the input time series used in this experiment.

Figure 1. Forward and inverse transform of a time series using DFT algorithm.

Figure 1. Forward and inverse transform of a time series using DFT algorithm.



Length is a power of two

The time series is 256 samples long. Although the DFT algorithm can accommodate time series of arbitrary lengths, I set the length of this time series to a power of two so that I can compare the results with results produced by an FFT algorithm later in the module.

(Recall that most FFT algorithms are restricted to input data lengths that are a power of two.)

The input time series

As you can see, the input time series consists of three concatenated pulses separated by blank spaces. The pulse on the leftmost end consists simply of some values that I entered into the time series to create a pulse with an interesting shape.

The middle pulse is a truncated sinusoid.

The rightmost pulse is a truncated square wave.

The objective

The objective of the experiment is to confirm that it is possible to transform this time series into the frequency domain using a forward Fourier transform, and then to recreate the time series by using an inverse Fourier transform to transform the complex spectrum back into the time domain.

The real part of the spectrum is symmetrical

The real part of the complex spectrum is shown in the second plot from the top in [Figure 1](#). It will become important later to note that the real part of the spectrum is symmetrical about the folding frequency near the center of the plot (*at the eighth tick mark*) .

Without attempting to explain why, I will simply tell you that the real part of the Fourier transform of a complex series whose imaginary part is all zeros (*like a typical sampled time series for real-world data*) is always symmetrical about the folding frequency.

The imaginary part of the spectrum is asymmetrical

The imaginary part of the complex spectrum is shown in the third plot from the top. Again, it will become important later to note that the imaginary part of the spectrum is asymmetrical about the folding frequency.

Once again, without attempting to explain why, the imaginary part of the Fourier transform of a complex series whose imaginary part is all zeros (*like a typical sampled time series for real-world data*) is always asymmetrical about the folding frequency.

The converse is also true

It is also true that the values of the imaginary part of the Fourier transform of a complex spectrum whose real part is symmetrical about the folding frequency and whose imaginary part is asymmetrical about the folding frequency will all be zero. I will take advantage of these facts later to simplify the computing and plotting process.

The amplitude spectrum

The amplitude spectrum is shown in the fourth plot down from the top. Recall from previous modules that the amplitude values are always positive, consisting of the square root of the sum of the squares of the real and imaginary parts.

The output time series

The output time series, produced by performing an inverse Fourier transform on the complex spectrum is shown in the bottom plot in [Figure 1](#). Compare the bottom plot to the top plot. As you can see, they are the same, demonstrating the reversible nature of the Fourier transform.

The program named Dsp035

I will discuss this program in fragments. A complete listing of the program is provided in [Listing 14](#) near the end of the module.

The beginning of the class named Dsp035

The beginning of the class for **Dsp035** , including the declaration of some variables and the creation of some array objects is shown in [Listing 1](#). This code is straightforward.

Listing 1. The beginning of the class named Dsp035.

```
class Dsp035 implements GraphIntf01{
    final double pi = Math.PI;

    int len = 256;

    double[] timeDataIn = new double[len];
    double[] realSpect = new double[len];
    double[] imagSpect = new double[len];
    double[] angle = new double[len]; //unused
    double[] magnitude = new double[len];
    double[] timeDataOut = new double[len];
```

Beginning of the constructor

The constructor begins in [Listing 2](#). The code in [Listing 2](#) creates the input time series data shown in the top plot of [Figure 1](#).

Listing 2. Beginning of the constructor.

```
public Dsp035(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
    //...
    //code deleted for brevity
    //...
    timeDataIn[254] = -80;
    timeDataIn[255] = -80;

    //Create raw data sinusoid
    for(int x = len/3;x < 3*len/4;x++){
        timeDataIn[x] = 80.0 * Math.sin(
                                2*pi*
(x)*1.0/20.0);
    }//end for loop
```

Note that I deleted much of the code from [Listing 2](#) for brevity. You can view the missing code in [Listing 14](#) near the end of the module.

Compute the complex spectrum

The code in [Listing 3](#) calls the static **transform** method of the **ForwardRealToComplex01** class to compute and save the complex spectrum of the time series shown in the top plot of [Figure 1](#).

Listing 3. Compute the complex spectrum.

```
ForwardRealToComplex01.transform(timeDataIn,  
                                   realSpect,  
                                   imagSpect,  
                                   angle,  
                                   magnitude,  
                                   zero,  
                                   0.0,  
                                   1.0);
```

The method parameters

I explained the **transform** method in the earlier module titled [Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#). The three middle plots in [Figure 1](#) are plots of the data returned in the arrays referred to by **realSpect** , **imagSpect** , and **magnitude** by the transform method.

The angle results returned by the transform program are not used in this module.

One of the parameters (*zero*) establishes that the first sample in the time series array referred to by **timeDataIn** represents the zero time origin.

The parameters also specify that the complex spectrum is to be computed at a set of equally spaced frequencies ranging from zero (*0.0*) to the sampling frequency (*1.0*) .

Perform the inverse Fourier transform

The code in [Listing 4](#) calls the static **inverseTransform** method of the **InverseComplexToReal01** class to perform an inverse Fourier transform on

the complex spectral data, producing the output time series shown in the bottom plot in [Figure 1](#).

Listing 4. Perform the inverse Fourier transform.

```
InverseComplexToReal01.inverseTransform(  
                                realSpect,  
                                imagSpect,  
                                timeDataOut);  
} //end constructor
```

I will explain the inverseTransform method later.

An object of the class **Dsp035**

[Listing 4](#) also signals the end of the constructor. Once the constructor has completed executing, an object of the **Dsp035** class exists. The array objects belonging to the object have been populated with the original time series, the complex spectrum of the original time series, and the output time series produced by performing an inverse Fourier transform on that complex spectrum. This data is ready for plotting.

The interface methods

All of the remaining code in **Dsp035** consists of the six methods necessary to satisfy the interface named **GraphIntfc01**. Those methods are required to provide data to the plotting program, as explained in earlier modules in this series.

If you have studied the earlier modules in this series, you probably don't want to hear any more about those methods, so I won't discuss them further. You can view the six interface methods in [Listing 14](#) near the end of the modules.

The **InverseComplexToReal01** class

The static method named **inverseTransform** performs a complex-to-real inverse discrete Fourier transform returning a real result only. In other words, the method transforms a complex input to a real output.

There are more efficient ways to write this method taking known symmetry and asymmetry conditions into account. However, I wrote the method the way that I did because I wanted it to mimic the behavior of an FFT algorithm. Therefore, the complex input must extend from zero to the sampling frequency.

The method does not implement an FFT algorithm. Rather, the **inverseTransform** method implements a straight-forward sampled-data version of the continuous inverse Fourier transform that is defined using integral calculus.

Parameters for the **inverseTransform** method

The parameters to the **inverseTransform** method are:

- `double[] realIn` - incoming real data
- `double[] imagIn` - incoming image data
- `double[] realOut` - outgoing real data

The method considers the data length to be **`realIn.length`** , and considers the computational time increment to be **`1.0/realIn.length`** .

Assumptions

The method returns a number of points equal to the data length. It assumes that the real input consists of positive frequency points for a symmetric real frequency function. That is, the real input is assumed to be symmetric about the folding frequency. The method does not test this assumption.

The method assumes that the imaginary input consists of positive frequency points for an asymmetric imaginary frequency function. That is, the imaginary input is assumed to be asymmetric about the folding frequency. Once again, the method does not test this assumption.

A real output

A symmetric real part and an asymmetric imaginary part guarantee that the imaginary output will be all zero values. Having made that assumption, the program makes no attempt to compute an imaginary output. If the assumptions described above are not valid, the results won't be valid.

The program was tested using JDK 1.8 under Windows 7.

Beginning of the `inverseTransform` method

The beginning of the class and the beginning of the static **`inverseTransform`** method is shown in [Listing 5](#).

Listing 5. Beginning of the class named `InverseComplexToReal01`.

Listing 5. Beginning of the class named InverseComplexToReal01.

```
public class InverseComplexToReal01{

    public static void inverseTransform(
                                double[] realIn,
                                double[] imagIn,
                                double[] realOut)
    {

        int dataLen = realIn.length;
        double delT = 1.0/realIn.length;
        double startTime = 0.0;
```

[Listing 5](#) declares and initializes some variables that will be used later.

The inverse transform computation

[Listing 6](#) contains a pair of nested for loops that perform the actual inverse transform computation.

Listing 6. The inverse transform computation.

Listing 6. The inverse transform computation.

```
//Outer loop iterates on time domain
// values.
for(int i=0; i < dataLen;i++){
    double time = startTime + i*delT;
    double real = 0;
    //Inner loop iterates on frequency
    // domain values.
    for(int j=0; j < dataLen; j++){
        real += realIn[j]*
                Math.cos(2*Math.PI*time*j)
                + imagIn[j]*
                Math.sin(2*Math.PI*time*j);
    }//end inner loop
    realOut[i] = real;
} //end outer loop
} //end inverseTransform

} //end class InverseComplexToReal01
```

If you have been studying the earlier modules in this series, you should be able to understand the code in [Listing 6](#) without further explanation. Pay particular attention to the comments that describe the two **for** loops.

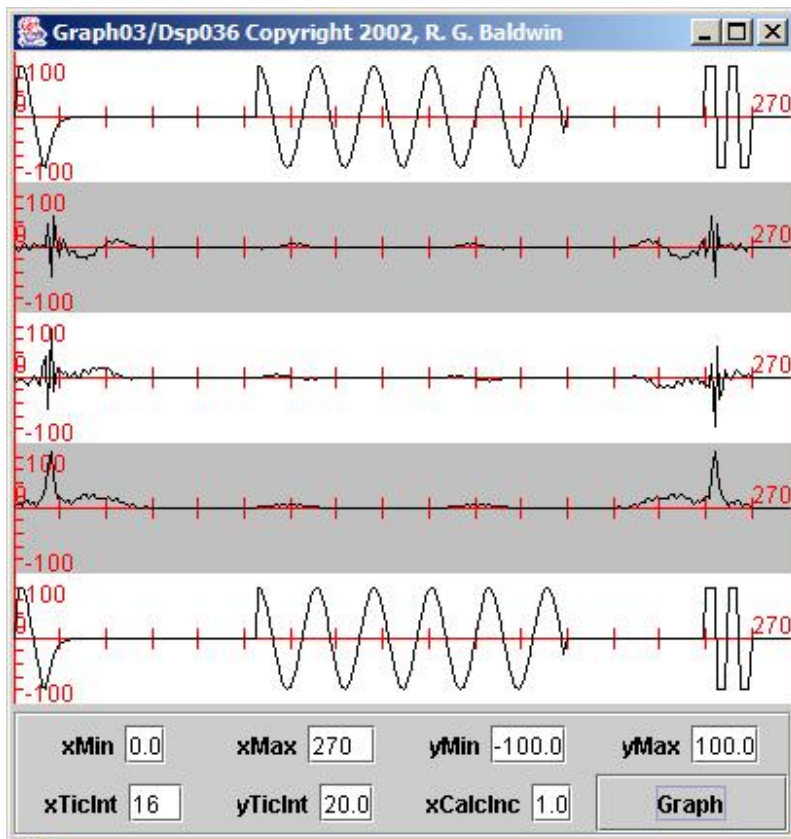
The program named Dsp036

The program named **Dsp036** replicates the behavior of the program named **Dsp035** , except that it uses an FFT algorithm to perform the inverse Fourier transform instead of using a DFT algorithm as in **Dsp035** .

The output from Dsp036

The output produced by running the program named **Dsp036** and plotting the output using the program named **Graph03** is shown in [Figure 2](#).

Figure 2. Forward and inverse transform of a time series using FFT algorithm.



Compare [Figure 2](#) with [Figure 1](#). The two should be identical. The program named **Dsp036** was designed to use an FFT algorithm for the inverse Fourier transform and to replicate the behavior of the program named **Dsp035**, which uses a DFT algorithm for the inverse Fourier transform. In addition, the same plotting parameters were used for both figures.

Some code from Dsp036

I'm only going to show you one short code fragment from the program named **Dsp036** . [Listing 7](#) shows the code that calls the methods to perform the forward and inverse Fourier transforms using the FFT algorithm. A complete listing of the program named **Dsp036** is shown in [Listing 16](#) near the end of the module.

Listing 7. Some code from Dsp036.

```
in    //Compute FFT of the time data and save it
      // the output arrays.
      ForwardRealToComplexFFT01.transform(
                                          timeDataIn,
                                          realSpect,
                                          imagSpect,
                                          angle,
                                          magnitude);

      //Compute inverse FFT of the spectral data
      InverseComplexToRealFFT01.

inverseTransform(
                                          realSpect,
                                          imagSpect,
                                          timeOut);
```

The **transform** method used to perform the forward Fourier transform in [Listing 7](#) was discussed in an earlier module, so I won't discuss it further here.

The inverse Fourier transform

The static **inverseTransform** method of the **InverseComplexToRealFFT01** class was used to perform the inverse Fourier transform in [Listing 7](#). You can view this method in [Listing 17](#) near the end of the module.

I'm not going to discuss this method in detail either, because it is very similar to the method named **InverseComplexToReal01** discussed earlier in conjunction with [Listing 4](#) and the listings following that one.

A couple of things to note

There are a couple of things, however, that I do want to point out.

The **transform** method and the **inverseTransform** method each call a method named **complexToComplex** to actually perform the Fourier transform. This method implements a classical FFT algorithm accepting complex input data and producing complex output data. The restriction of real-to-complex and complex-to-real is imposed in this program by the methods named **transform** and **inverseTransform**.

*(The method named **complexToComplex** is also suitable for use if you have a need to perform complex-to-complex Fourier transforms.)*

The signature of the complexToComplex method

The signature for the **complexToComplex** method is shown in [Figure 3](#).

Figure 3. The signature of the `complexToComplex` method.

```
public static void complexToComplex(  
    int sign,  
    int len,  
    double real[],  
    double imag[])  
{
```

The **`complexToComplex`** method can be used to perform either a forward or an inverse transform. The value of the first parameter determines whether the method performs a forward or an inverse Fourier transform.

The first parameter of the `complexToComplex` method

A value of +1 for the first parameter causes the **`complexToComplex`** method to perform a forward Fourier transform.

A value of -1 for the first parameter causes the **`complexToComplex`** method to perform an inverse Fourier transform.

The forward transform

Although I didn't include the code in this module, (*because it was shown in an earlier module*) , the **`transform`** method in [Figure 7](#) passes a value of +1 to the **`complexToComplex`** method to cause it to perform a forward Fourier transform.

The inverse transform

Similarly, the **`inverseTransform`** method shown in [Listing 17](#) passes a value of -1 to the **`complexToComplex`** method to cause it to perform an inverse

Fourier transform.

FFT and DFT produce equivalent results

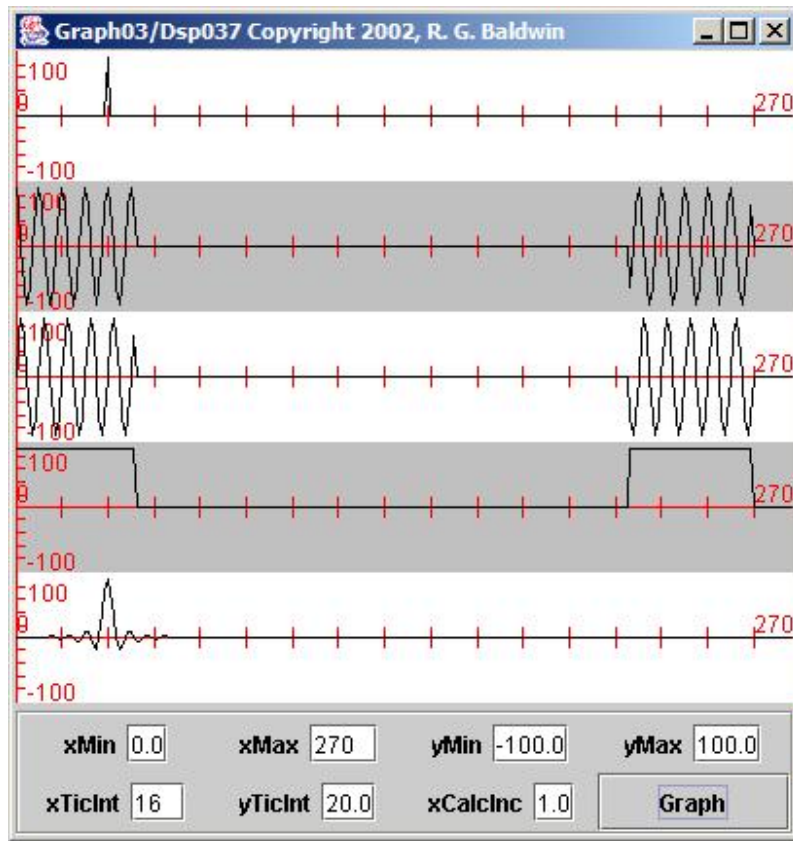
As evidenced in [Figure 1](#) and [Figure 2](#), the program named **Dsp035**, which uses a DFT algorithm, produces the same results as the program named **Dsp036**, which uses an FFT algorithm. However, if you were to put a timer on each of the programs, you would find that **Dsp036** runs faster due to the improved speed of the FFT algorithm over the DFT algorithm.

Using a Fourier transform to perform frequency filtering

The program named **Dsp037** illustrates frequency filtering accomplished by modifying the complex spectrum in the frequency domain and then performing an inverse Fourier transform on the modified frequency-domain data. The results are shown in [Figure 4](#).

Figure 4. Filtering in the frequency domain.

Figure 4. Filtering in the frequency domain.



Operation of the program

The program begins by using an FFT algorithm to perform a forward Fourier transform on a single impulse in the time domain.

(A DFT algorithm could have been used equally as well, but it would have been slower.)

The impulse is shown as the input time series in the topmost plot in [Figure 4](#).

(Although I didn't show the complex spectrum of the impulse, we know that the magnitude of the spectrum of an impulse is constant across all frequencies. In other words, the magnitude spectrum of an impulse is a flat line from zero to the sampling frequency and above.)

Modify the complex spectrum

Then the program eliminates all energy between one-sixth and five-sixths of the sampling frequency by setting the real and imaginary parts of the FFT output to zero.

The second, third, and fourth plots in [Figure 4](#) show the real part, imaginary part, and amplitude respectively of the modified complex spectrum.

(The two boxes in the fourth plot in [Figure 4](#) show what's left of the spectral energy after the energy in the middle of the band has been eliminated.)

The folding frequency in these three plots is near the center of the plot at the eighth tick mark.

The plotting format

The input data length was 256 samples. All but one of the input data values was set to zero resulting in a single impulse in the input time series near the second tick mark in [Figure 4](#).

(The real and complex parts of the frequency spectrum were computed at 256 frequencies between zero and the sampling frequency.)

There were 256 values plotted horizontally in each separate plot. Once again, to make it easier to view the plots on the rightmost end, I plotted the values on a grid that is 270 units wide. This leaves some blank space on the rightmost end to contain the numbers, thus preventing the numbers from being mixed in with the plotted values. The last actual data value coincides with the rightmost tick mark on each plot.

Perform an inverse Fourier transform

After modifying the complex spectrum as described above, the program performs an inverse Fourier transform on the modified complex spectrum to produce the filtered impulse.

The filtered impulse

The filtered impulse is shown as the bottom plot in [Figure 4](#). As you can see, the pulse is smeared out in time relative to the input pulse in the top plot. This is the typical result of reducing the bandwidth of a pulse.

(This particular modification of the complex spectrum resulted in a filtered pulse that has the waveform of a $\text{SIN}(X)/X$ function. A different modification of the complex spectrum would have resulted in a filtered pulse with a different waveform.)

This example also illustrates one of the miracles of digital signal processing. Energy appears in the output before it occurs in the input. Obviously that is not possible in the real world of analog

systems, but many things are possible in the digital world that are not possible in the real world.)

Beginning of the class for Dsp037

[Listing 8](#) shows the beginning of the class definition for the program named Dsp037.

Listing 8. Beginning of the class for Dsp037.

```
class Dsp037 implements GraphIntf01{
    final double pi = Math.PI;

    int len = 256;

    double[] timeDataIn = new double[len];
    double[] realSpect = new double[len];
    double[] imagSpect = new double[len];
    double[] angle = new double[len]; //unused
    double[] magnitude = new double[len];
    double[] timeOut = new double[len];
```

[Listing 8](#) simply declares and initializes some variables that will be used later.

Beginning of the constructor

The constructor begins in [Listing 9](#).

Listing 9. Beginning of the constructor.

```
public Dsp037(){//constructor  
    timeDataIn[32] = 90;
```

[Listing 9](#) creates the raw pulse data shown in the topmost plot in [Figure 4](#).

When the array object referred to by **timeDataIn** is created, the values of all array elements are set to zero by default. [Listing 9](#) modifies one of the elements to have a value of 90. This results in a single impulse at an index of 32.

Compute the Fourier transform

Continuing with the constructor, the code in [Listing 10](#) uses an FFT algorithm in the method named **transform** (*discussed earlier*) to compute the Fourier transform of the impulse.

Listing 10. Compute the Fourier transform.

Listing 10. Compute the Fourier transform.

```
in    //Compute FFT of the time data and save it
      // the output arrays.
      ForwardRealToComplexFFT01.transform(
                                          timeDataIn,
                                          realSpect,
                                          imagSpect,
                                          angle,
                                          magnitude);
```

The results of the Fourier transform are stored in the array objects referred to by **realSpect** , **imagSpect** , and **magnitude** .

(The phase angle is also computed but is of no interest in this example.)

Apply the filter to the frequency data

[Listing 11](#) applies the filter by setting sample values in a portion of the real and imaginary parts of the complex spectrum to zero.

Listing 11. Apply the filter to the frequency data.

Listing 11. Apply the filter to the frequency data.

```
for(int cnt = len/6;cnt < 5*len/6;cnt++){  
    realSpect[cnt] = 0.0;  
    imagSpect[cnt] = 0.0;  
} //end for loop
```

This code eliminates all energy between one-sixth and five-sixths of the sampling frequency. The modified data for the real and imaginary parts of the complex spectrum are shown in the second and third plots in [Figure 4](#).

Re-compute the magnitude

[Listing 12](#) re-computes the magnitude values for the modified real and imaginary values of the complex spectrum.

Listing 12. Re-compute the magnitude.

```
//Recompute the magnitude based on the  
// modified real and imaginary spectra.  
for(int cnt = 0;cnt < len;cnt++){  
    magnitude[cnt] =  
        (Math.sqrt(  
            realSpect[cnt]*realSpect[cnt]  
            +  
            imagSpect[cnt]*imagSpect[cnt])/len);  
} //end for loop
```

The modified data for the amplitude of the complex spectrum are shown in the fourth plot in [Figure 4](#).

Compute the inverse Fourier transform

[Listing 13](#) uses the `inverseTransform` method to compute the inverse Fourier transform of the modified complex spectrum stored in `realSpect` and `imagSpect`. The results of the inverse transform are stored in `timeOut`.

Listing 13. Compute the inverse Fourier transform.

```
        InverseComplexToRealFFT01.inverseTransform(  
realSpect,  
imagSpect,  
timeOut);  
    }//end constructor
```

The results of the inverse transform are shown in the bottom plot in [Figure 4](#).

[Listing 13](#) also signals the end of the constructor.

Display the results

Once the constructor returns, all of the data that is to be plotted has been stored in the various array objects. The remaining code in the program consists of the definition of the six methods required by the interface named

GraphIntfc01. These methods are required to make it possible to use the program named Graph03 to plot the results as shown in [Figure 4](#).

I have discussed these methods on numerous previous occasions, and won't repeat that discussion here.

One more example, Dsp038

[Figure 5](#) illustrates one more example of performing frequency filtering by modifying the complex spectrum and then performing an inverse transform on the modified spectrum.

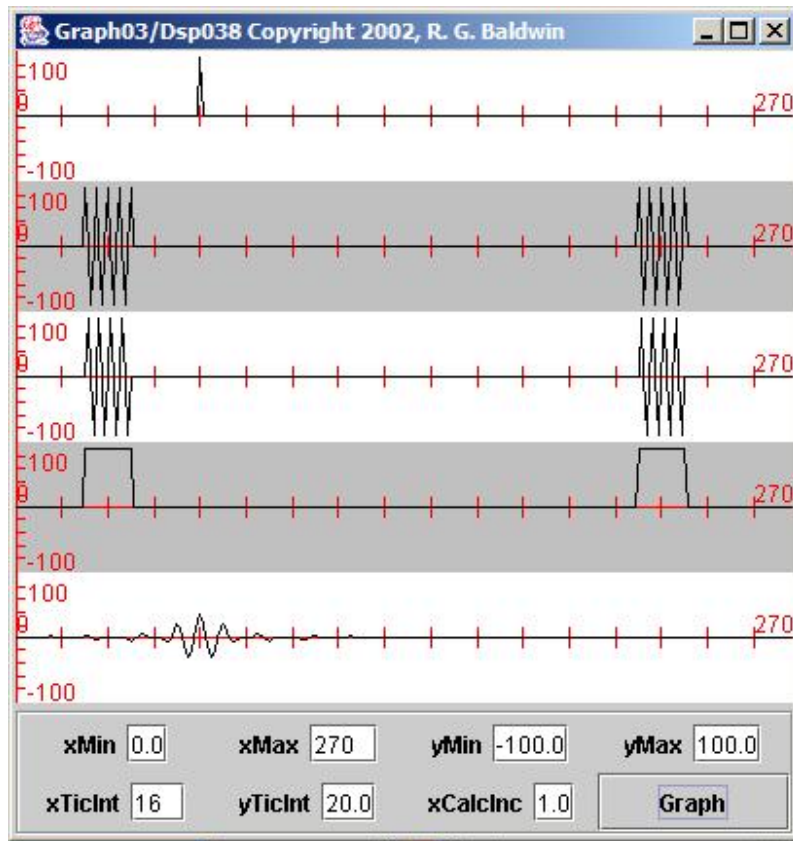
While discussing the program named **Dsp037**, I told you that performing a different modification on the complex spectrum would result in a different waveform for the filtered impulse. The program named **Dsp038** applies a different modification to the complex spectrum, but is otherwise the same as Dsp037.

(Because of the similarity of the two programs, I won't discuss the code in Dsp038. You can view that code in [Listing 19](#) near the end of the module.)

[Figure 5](#) shows the output produced by the program named **Dsp038**.

Figure 5. Filtering in the frequency domain.

Figure 5. Filtering in the frequency domain.



Compare the results

The basic plotting format of [Figure 5](#) is the same as [Figure 4](#).

Compare [Figure 5](#) with [Figure 4](#)

The first difference to note between the two figures is that I moved the impulse in the input time series in the topmost plot sixteen samples further to the right in **Dsp038**.

(This has no impact on the final result, which you can verify by modifying the program to move the impulse to a different position and then compiling and running the modified program.)

Compare the bandwidth of the pass band

The second difference to note is shown in the modified amplitude spectrum in the fourth plot in the two figures. The bandwidth of the pass band is significantly narrower in [Figure 5](#) than in [Figure 4](#). Also, the pass band in [Figure 4](#) extends all the way down to zero frequency, while [Figure 5](#) eliminates all energy below a frequency of three thirty-seconds of the sampling frequency.

Waveforms of filtered impulse

Finally, note the waveforms of the two filtered impulses. The overall amplitude of the filtered impulse in [Figure 5](#) is less than in [Figure 4](#), simply because it contains less total energy. In addition, the filtered impulse in [Figure 5](#) is broader than the filtered impulse in [Figure 4](#). This is because it has a narrower bandwidth.

(Pulses that are narrow in terms of time duration require a wider bandwidth than pulses that have a longer time duration. The time duration of the pulse tends to be inversely related to the required bandwidth for the pulse.)

Run the programs

I encourage you to copy, compile, and run the programs provided in this module. Experiment with them, making changes and observing the results of

your changes.

Create more complex experiments. For example, use more complex input time series when experimenting with frequency filtering. Apply different modifications to the complex spectrum when experimenting with frequency filtering.

Most of all enjoy yourself and learn something in the process.

Summary

This module illustrates and explains forward and inverse Fourier transforms using both DFT and FFT algorithms.

The module also illustrates and explains the implementation of frequency filtering by modifying the complex spectrum in the frequency domain and then transforming the modified complex spectrum back into the time domain.

Complete program listings

Complete listings of the programs discussed in this module are provided below.

Listings for other programs mentioned in the module, such as **Graph03** and **Graph06** , are provided in other modules. Those modules are identified in the text of this module.

Listing 14. Dsp035.java.

```
import java.util.*;

class Dsp035 implements GraphIntf01{
```

Listing 14. Dsp035.java.

```
final double pi = Math.PI;

int len = 256;

double[] timeDataIn = new double[len];
double[] realSpect = new double[len];
double[] imagSpect = new double[len];
double[] angle = new double[len]; //unused
double[] magnitude = new double[len];
double[] timeDataOut = new double[len];
int zero = 0;

public Dsp035(){//constructor

    //Create the raw data pulses
    timeDataIn[0] = 0;
    timeDataIn[1] = 50;
    timeDataIn[2] = 75;
    timeDataIn[3] = 80;
    timeDataIn[4] = 75;
    timeDataIn[5] = 50;
    timeDataIn[6] = 25;
    timeDataIn[7] = 0;
    timeDataIn[8] = -25;
    timeDataIn[9] = -50;
    timeDataIn[10] = -75;
    timeDataIn[11] = -80;
    timeDataIn[12] = -60;
    timeDataIn[13] = -40;
    timeDataIn[14] = -26;
    timeDataIn[15] = -17;
    timeDataIn[16] = -11;
    timeDataIn[17] = -8;
    timeDataIn[18] = -5;
    timeDataIn[19] = -3;
    timeDataIn[20] = -2;
```

Listing 14. Dsp035.java.

```
timeDataIn[21] = -1;

timeDataIn[240] = 80;
timeDataIn[241] = 80;
timeDataIn[242] = 80;
timeDataIn[243] = 80;
timeDataIn[244] = -80;
timeDataIn[245] = -80;
timeDataIn[246] = -80;
timeDataIn[247] = -80;
timeDataIn[248] = 80;
timeDataIn[249] = 80;
timeDataIn[250] = 80;
timeDataIn[251] = 80;
timeDataIn[252] = -80;
timeDataIn[253] = -80;
timeDataIn[254] = -80;
timeDataIn[255] = -80;

//Create raw data sinusoid
for(int x = len/3;x < 3*len/4;x++){
    timeDataIn[x] = 80.0 * Math.sin(
        2*pi*
(x)*1.0/20.0);
} //end for loop

//Compute DFT of the time data and save it
in
// the output arrays.
ForwardRealToComplex01.transform(timeDataIn,
                                realSpect,
                                imagSpect,
                                angle,
                                magnitude,
                                zero,
                                0.0,
```

Listing 14. Dsp035.java.

```
1.0);

//Compute inverse DFT of spectral data and
// save output time data in output array
InverseComplexToReal01.inverseTransform(
    realSpect,
    imagSpect,
    timeDataOut);
} //end constructor

//-----
-//
//The following six methods are required by
the
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of curves to plot. Must not
    // exceed 5.
    return 5;
} //end getNmbr
//-----
-//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > timeDataIn.length-1)
{
    return 0;
    }else{
    return timeDataIn[index];
    } //end else
} //end function
//-----
-//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > realSpect.length-1){
```

Listing 14. Dsp035.java.

```
        return 0;
    }else{
        //scale for convenient viewing
        return 5*realSpect[index];
    }//end else
} //end function
//-----
-//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imagSpect.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return 5*imagSpect[index];
    }//end else
} //end function
//-----
-//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > magnitude.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return 5*magnitude[index];
    }//end else
} //end function
//-----
-//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > timeDataOut.length-1)
{
    return 0;
```

Listing 14. Dsp035.java.

```
        }else{
            return timeDataOut[index];
        }//end else
    }//end function

} //end sample class Dsp035
```

Listing 15. InverseComplexToReal01.hava.

```
/*File InverseComplexToReal01.java
Copyright 2004, R.G.Baldwin
Rev 5/24/04
```

Although there are more efficient ways to write this program, it was written the way it was to mimic the behavior of an FFT algorithm. Therefore, the complex input must extend from zero to the sampling frequency.

The static method named `inverseTransform` performs a complex to real inverse discrete Fourier transform returning a real result only. In other words, the method transforms a complex input to a real output.

Does not implement the FFT algorithm. Implements a straight-forward sampled-data version of the continuous inverse Fourier transform defined using integral calculus.

Listing 15. InverseComplexToReal01.hava.

The parameters are:

double[] realIn - incoming real data
double[] imagIn - incoming imag data
double[] realOut - outgoing real data

Considers the data length to be
realIn.length
Computational time increment is
1.0/realIn.length

Returns a number of points equal to the data
length.

Assumes real input consists of positive
frequency points for a symmetric real frequency
function. That is, the real input is assumed to
be symmetric about the folding frequency. Does
not test this assumption.

Assumes imaginary input consists of positive
frequency points for an asymmetric imaginary
frequency function. That is, the imaginary input
is assumed to be asymmetric about the
folding frequency. Does not test this
assumption.

The assumption of a symmetric real part and an
asymmetric imaginary part guarantees that the
imaginary output would be all zero if it were to
be computed. Thus the program makes no attempt
to compute an imaginary output.

Tested using J2SE v1.4.2 under WinXP.

*****/

Listing 15. InverseComplexToReal01.hava.

```
public class InverseComplexToReal01{

    public static void inverseTransform(
                                double[] realIn,
                                double[] imagIn,
                                double[] realOut){

        int dataLen = realIn.length;
        double delT = 1.0/realIn.length;
        double startTime = 0.0;
        //Outer loop iterates on time domain
        // values.
        for(int i=0; i < dataLen;i++){
            double time = startTime + i*delT;
            double real = 0;
            //Inner loop iterates on frequency
            // domain values.
            for(int j=0; j < dataLen; j++){
                real += realIn[j]*
                        Math.cos(2*Math.PI*time*j)
                        + imagIn[j]*
                        Math.sin(2*Math.PI*time*j);
            }//end inner loop
            realOut[i] = real;
        }//end outer loop
    }//end inverseTransform

}//end class InverseComplexToReal01
```

Listing 16. Dsp036.java.

Listing 16. Dsp036.java.

```
/* File Dsp036.java  
Copyright 2004, R.G.Baldwin  
Revised 5/24/04
```

Illustrates forward and inverse Fourier transforms using FFT algorithms.

Performs spectral analysis on a time series consisting of pulses and a sinusoid.

Passes resulting real and complex parts to inverse Fourier transform program to reconstruct the original time series.

Run with Graph03.

Tested using J2SE 1.4.2 under WinXP.

```
*****/  
import java.util.*;
```

```
class Dsp036 implements GraphIntf01{  
    final double pi = Math.PI;  
  
    int len = 256;  
  
    double[] timeDataIn = new double[len];  
    double[] realSpect = new double[len];  
    double[] imagSpect = new double[len];  
    double[] angle = new double[len]; //unused  
    double[] magnitude = new double[len];  
    double[] timeOut = new double[len];  
  
    public Dsp036(){ //constructor  
  
        //Create the raw data pulses  
        timeDataIn[0] = 0;
```

Listing 16. Dsp036.java.

```
timeDataIn[1] = 50;  
timeDataIn[2] = 75;  
timeDataIn[3] = 80;  
timeDataIn[4] = 75;  
timeDataIn[5] = 50;  
timeDataIn[6] = 25;  
timeDataIn[7] = 0;  
timeDataIn[8] = -25;  
timeDataIn[9] = -50;  
timeDataIn[10] = -75;  
timeDataIn[11] = -80;  
timeDataIn[12] = -60;  
timeDataIn[13] = -40;  
timeDataIn[14] = -26;  
timeDataIn[15] = -17;  
timeDataIn[16] = -11;  
timeDataIn[17] = -8;  
timeDataIn[18] = -5;  
timeDataIn[19] = -3;  
timeDataIn[20] = -2;  
timeDataIn[21] = -1;  
  
timeDataIn[240] = 80;  
timeDataIn[241] = 80;  
timeDataIn[242] = 80;  
timeDataIn[243] = 80;  
timeDataIn[244] = -80;  
timeDataIn[245] = -80;  
timeDataIn[246] = -80;  
timeDataIn[247] = -80;  
  
timeDataIn[248] = 80;  
timeDataIn[249] = 80;  
timeDataIn[250] = 80;  
timeDataIn[251] = 80;
```

Listing 16. Dsp036.java.

```
timeDataIn[252] = -80;
timeDataIn[253] = -80;
timeDataIn[254] = -80;
timeDataIn[255] = -80;

//Create raw data sinusoid
for(int x = len/3;x < 3*len/4;x++){
    timeDataIn[x] = 80.0 * Math.sin(
        2*pi*(x)*1.0/20.0);
}

//end for loop

//Compute FFT of the time data and save it in
// the output arrays.
ForwardRealToComplexFFT01.transform(
    timeDataIn,
    realSpect,
    imagSpect,
    angle,
    magnitude);

//Compute inverse FFT of spectral data
InverseComplexToRealFFT01.
    inverseTransform(
        realSpect,
        imagSpect,
        timeOut);

}

//end constructor

//-----//
//The following six methods are required by the
// interface named GraphIntf01.
public int getNmbr(){
    //Return number of curves to plot. Must not
    // exceed 5.
    return 5;
}

//end getNmbr
```

Listing 16. Dsp036.java.

```
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > timeDataIn.length-1){
        return 0;
    }else{
        return timeDataIn[index];
    }//end else
}//end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > realSpect.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return 5*realSpect[index]/len;
    }//end else
}//end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imagSpect.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return 5*imagSpect[index]/len;
    }//end else
}//end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude.length-1){
        return 0;
    }else{
```

Listing 16. Dsp036.java.

```
        //scale for convenient viewing
        return 5*magnitude[index];
    }//end else
} //end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > timeOut.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return timeOut[index]/len;
    } //end else
} //end function

} //end sample class Dsp036
```

Listing 17. InverseComplexToRealFFT01.java,

```
/*File InverseComplexToRealFFT01.java
Copyright 2004, R.G.Baldwin
Rev 5/24/04
```

The static method named `inverseTransform` performs a complex to real Fourier transform using a complex-to-complex FFT algorithm. A specific parameter is passed to the FFT algorithm that causes this to be an inverse Fourier transform.

Listing 17. InverseComplexToRealFFT01.java,

See InverseComplexToReal01 for a version that does not use an FFT algorithm but uses a DFT algorithm instead.

Incoming parameters are:

- double[] realIn - incoming real data
- double[] imagIn - incoming imaginary data
- double[] realOut - outgoing real data

Requires spectral input data extending from zero to the sampling frequency.

Assumes real input consists of positive frequency points for a symmetric real frequency function. That is, the real input is assumed to be symmetric about the folding frequency. Does not test this assumption.

Assumes imaginary input consists of positive frequency points for an asymmetric imaginary frequency function. That is, the imaginary input is assumed to be asymmetric about the folding frequency. Does not test this assumption.

The assumption of a symmetric real part and an asymmetric imaginary part guarantees that the imaginary output is all zeros. Thus, the program does not return an imaginary output. Does not test the assumption that the imaginary is all zeros.

CAUTION: THE INCOMING DATA LENGTH MUST BE A POWER OF TWO. OTHERWISE, THIS PROGRAM WILL FAIL TO RUN PROPERLY.

Listing 17. InverseComplexToRealFFT01.java,

Returns a number of points equal to the incoming data length. Those points are uniformly distributed beginning at zero.

*****/

```
public class InverseComplexToRealFFT01{

    public static void inverseTransform(
                                double[] realIn,
                                double[] imagIn,
                                double[] realOut){
        double pi = Math.PI;//for convenience
        int dataLen = realIn.length;
        double[] imagOut = new double[dataLen];
        //The complexToComplex FFT method does an
        // in-place transform causing the output
        // complex data to be stored in the arrays
        // containing the input complex data.
        // Therefore, it is necessary to copy the
        // input data into the output arrays before
        // passing them to the FFT algorithm.
        System.arraycopy(realIn,0,realOut,0,dataLen);
        System.arraycopy(imagIn,0,imagOut,0,dataLen);

        //Perform the spectral analysis. The results
        // are stored in realOut and imagOut. Note
        // that the -1 value for the first
        // parameter causes the transform to be an
        // inverse transform. A +1 value would cause
        // it to be a forward transform.
        complexToComplex(-1,dataLen,realOut,imagOut);

    }//end inverseTransform method
    //-----//
```

Listing 17. InverseComplexToRealFFT01.java,

```
//This method computes a complex-to-complex
// FFT. The value of sign must be 1 for a
// forward FFT and -1 for an inverse FFT.
public static void complexToComplex(
                                int sign,
                                int len,
                                double real[],
                                double imag[]){

    double scale = 1.0;
    //Reorder the input data into reverse binary
    // order.
    int i,j;
    for (i=j=0; i < len; ++i) {
        if (j>=i) {
            double tempr = real[j]*scale;
            double tempi = imag[j]*scale;
            real[j] = real[i]*scale;
            imag[j] = imag[i]*scale;
            real[i] = tempr;
            imag[i] = tempi;
        }//end if
        int m = len/2;
        while (m>=1 && j>=m) {
            j -= m;
            m /= 2;
        }//end while loop
        j += m;
    }//end for loop

    //Input data has been reordered.
    int stage = 0;
    int maxSpectraForStage,stepSize;
    //Loop once for each stage in the spectral
    // recombination process.
    for(maxSpectraForStage = 1,
        stepSize = 2*maxSpectraForStage;
```

Listing 17. InverseComplexToRealFFT01.java,

```
        maxSpectraForStage < len;
        maxSpectraForStage = stepSize,
        stepSize = 2*maxSpectraForStage){
double deltaAngle =
        sign*Math.PI/maxSpectraForStage;
//Loop once for each individual spectra
for (int spectraCnt = 0;
        spectraCnt < maxSpectraForStage;
        ++spectraCnt){
    double angle = spectraCnt*deltaAngle;
    double realCorrection = Math.cos(angle);
    double imagCorrection = Math.sin(angle);

    int right = 0;
    for (int left = spectraCnt;
            left < len;left += stepSize){
        right = left + maxSpectraForStage;
        double tempReal =
            realCorrection*real[right]
            - imagCorrection*imag[right];
        double tempImag =
            realCorrection*imag[right]
            + imagCorrection*real[right];
        real[right] = real[left]-tempReal;
        imag[right] = imag[left]-tempImag;
        real[left] += tempReal;
        imag[left] += tempImag;
    }//end for loop
    }//end for loop for individual spectra
    maxSpectraForStage = stepSize;
} //end for loop for stages
} //end complexToComplex method

} //end class InverseComplexToRealFFT01
```

Listing 18. Dsp037.java.

```
/* File Dsp037.java  
Copyright 2004, R.G.Baldwin  
Revised 5/24/04
```

Illustrates filtering in the frequency domain.
Performs FFT on an impulse. Eliminates all
energy between one-sixth and five-sixths of the
sampling frequency by modifying the real and
imaginary parts of the FFT output. Then performs
inverse FFT to produce the filtered impulse.

Run with Graph03.

Tested using J2SE 1.4.2 under WinXP.

```
*****/
```

```
import java.util.*;
```

```
class Dsp037 implements GraphIntf01{  
    final double pi = Math.PI;
```

```
    int len = 256;
```

```
    double[] timeDataIn = new double[len];  
    double[] realSpect = new double[len];  
    double[] imagSpect = new double[len];  
    double[] angle = new double[len]; //unused  
    double[] magnitude = new double[len];  
    double[] timeOut = new double[len];
```

```
    public Dsp037(){ //constructor
```

```
        //Create the raw data pulse  
        timeDataIn[32] = 90;
```

```
        //Compute FFT of the time data and save it in
```

Listing 18. Dsp037.java.

```
// the output arrays.
ForwardRealToComplexFFT01.transform(
                                timeDataIn,
                                realSpect,
                                imagSpect,
                                angle,
                                magnitude);

//Apply the frequency filter eliminating all
// energy between one-sixth and five-sixths
// of the sampling frequency by modifying the
// real and imaginary parts of the spectrum.
for(int cnt = len/6;cnt < 5*len/6;cnt++){
    realSpect[cnt] = 0.0;
    imagSpect[cnt] = 0.0;
}//end for loop

//Recompute the magnitude based on the
// modified real and imaginary spectra.
for(int cnt = 0;cnt < len;cnt++){
    magnitude[cnt] =
        (Math.sqrt(
            realSpect[cnt]*realSpect[cnt]
            + imagSpect[cnt]*imagSpect[cnt])/len);
}//end for loop

//Compute inverse FFT of modified spectral
// data.
InverseComplexToRealFFT01.inverseTransform(
                                realSpect,
                                imagSpect,
                                timeOut);

}//end constructor

//-----//
//The following six methods are required by the
```

Listing 18. Dsp037.java.

```
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of curves to plot.  Must not
    // exceed 5.
    return 5;
} //end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > timeDataIn.length-1){
        return 0;
    }else{
        return timeDataIn[index];
    } //end else
} //end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > realSpect.length-1){
        return 0;
    }else{
        return realSpect[index];
    } //end else
} //end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imagSpect.length-1){
        return 0;
    }else{
        return imagSpect[index];
    } //end else
} //end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
```

Listing 18. Dsp037.java.

```
        if(index < 0 ||
           index > magnitude.length-1){
            return 0;
        }else{
            //scale for convenient viewing
            return len*magnitude[index];
        }//end else
    }//end function
    //-----//
    public double f5(double x){
        int index = (int)Math.round(x);
        if(index < 0 ||
           index > timeOut.length-1){
            return 0;
        }else{
            //scale for convenient viewing
            return 3.0*timeOut[index]/len;
        }//end else
    }//end function

}//end sample class Dsp037
```

Listing 19. Dsp038.java.

```
/* File Dsp038.java
Copyright 2004, R.G.Baldwin
Revised 5/24/04
```

Illustrates filtering in the frequency domain.

Listing 19. Dsp038.java.

Performs FFT on an impulse. Modifies the complex spectrum. Then performs inverse FFT to produce the filtered impulse.

Run with Graph03.

Tested using J2SE 1.4.2 under WinXP.

```

*****/
import java.util.*;

```

```
class Dsp038 implements GraphIntfc01{
    final double pi = Math.PI;
```

```
int len = 256;
```

```
double[] timeDataIn = new double[len];
double[] realSpect = new double[len];
double[] imagSpect = new double[len];
double[] angle = new double[len]; //unused
double[] magnitude = new double[len];
double[] timeOut = new double[len];
```

```
public Dsp038(){//constructor
```

```
//Create the raw data pulse
timeDataIn[64] = 90;
```

```
//Compute FFT of the time data and save it in
// the output arrays.
```

```
ForwardRealToComplexFFT01.transform(
    timeDataIn,
    realSpect,
    imagSpect,
    angle,
    magnitude);
```


Listing 19. Dsp038.java.

```
//Apply the frequency filter.
for(int cnt = 0;cnt <= len/2;cnt++){
    if(cnt < 3*len/32){
        realSpect[cnt] = 0;
        imagSpect[cnt] = 0;
    }//end if

    if(cnt > 5*len/32){
        realSpect[cnt] = 0;
        imagSpect[cnt] = 0;
    }//end if

    //Fold complex spectral data
    if(cnt > 0){
        realSpect[len - cnt] = realSpect[cnt];
    }//end if
    if(cnt > 0){
        imagSpect[len - cnt] = -imagSpect[cnt];
    }//end if
} //end for loop

//Recompute the magnitude based on the
// modified real and imaginary spectra.
for(int cnt = 0;cnt < len;cnt++){
    magnitude[cnt] =
        (Math.sqrt(
            realSpect[cnt]*realSpect[cnt]
            + imagSpect[cnt]*imagSpect[cnt])/len);
} //end for loop

//Compute inverse FFT of modified spectral
// data.
InverseComplexToRealFFT01.inverseTransform(
    realSpect,
    imagSpect,
    timeOut);
```

Listing 19. Dsp038.java.

```
//end constructor

//-----//
//The following six methods are required by the
// interface named GraphIntfc01.
public int getNmbr(){
    //Return number of curves to plot. Must not
    // exceed 5.
    return 5;
}//end getNmbr
//-----//
public double f1(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > timeDataIn.length-1){
        return 0;
    }else{
        return timeDataIn[index];
    }//end else
}//end function
//-----//
public double f2(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > realSpect.length-1){
        return 0;
    }else{
        return realSpect[index];
    }//end else
}//end function
//-----//
public double f3(double x){
    int index = (int)Math.round(x);
    if(index < 0 || index > imagSpect.length-1){
        return 0;
    }else{
        return imagSpect[index];
    }//end else
}
```

Listing 19. Dsp038.java.

```
//end function
//-----//
public double f4(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > magnitude.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return len*magnitude[index];
    }//end else
}//end function
//-----//
public double f5(double x){
    int index = (int)Math.round(x);
    if(index < 0 ||
        index > timeOut.length-1){
        return 0;
    }else{
        //scale for convenient viewing
        return 3.0*timeOut[index]/len;
    }//end else
}//end function

}//end sample class Dsp038
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1485-Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain
- File: Java1485
- Published: 11/16/04

Baldwin illustrates and explains forward and inverse Fourier transforms using both DFT and FFT algorithms. He also illustrates and explains the implementation of frequency filtering by modifying the complex spectrum in the frequency domain and transforming the modified complex spectrum back into the time domain.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1486-Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm

Baldwin explains the underlying signal processing concepts that make the Fast Fourier Transform (FFT) algorithm possible.

Revised: Mon Oct 19 13:36:25 CDT 2015

This page is included in the following book: [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General discussion](#)
 - [A general-purpose transform](#)
 - [Transforming from space domain to wave number domain](#)
 - [Two-dimensional Fourier transforms](#)
 - [Wave-number response to seismic waves](#)
 - [A general purpose mathematical transform](#)
 - [Fourier transform images](#)
 - [Will discuss underlying concepts](#)
 - [A linear transform](#)
 - [Output display of the FFT applet](#)
 - [Back to the concept of the linear transform](#)
 - [A mirror-image pulse](#)
 - [Now add the two input series](#)

- [Single sample real pulse \(impulse\) with a delay](#)
- [Equations to describe the real and imaginary parts of the transform](#)
- [A sample program](#)
 - [Separate processes in an FFT algorithm](#)
 - [How the processes are implemented](#)
 - [Three cases are examined](#)
 - [The program named Fft02](#)
 - [Instantiate a Transform object](#)
 - [The class named Transform](#)
 - [Performing the transform](#)
 - [The correctAndRecombine method](#)
 - [Back to the main method](#)
 - [The graphic form of Case A](#)
 - [The numeric output for Case A](#)
 - [Case B code](#)
 - [Case B in graphical form](#)
 - [Case B output in numeric form](#)
 - [Case C code](#)
 - [The graphic form of Case C](#)
 - [Case C output in numeric form](#)
 - [The display method](#)
- [Run the program](#)
- [Summary](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

Programming in Java doesn't have to be dull and boring. In fact, it's possible to have a lot of fun while programming in Java. This module was taken from a series that concentrates on having fun while programming in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Transform of pulse with negative slope.
- [Figure 2.](#) Transform of pulse with positive slope.
- [Figure 3.](#) Transform of the sum of two pulses.
- [Figure 4.](#) Transform of an impulse with no shift.
- [Figure 5.](#) Transform of an impulse with a shift equal to one sample interval and a negative value.
- [Figure 6.](#) Transform of an impulse with a shift equal to two sample intervals and a positive value.
- [Figure 7.](#) Transform of an impulse with a shift equal to four sample intervals and a positive value.
- [Figure 8.](#) Transform of a complex impulse with a shift equal to two sample intervals.
- [Figure 9.](#) Case A. Transform of a real sample with two non-zero values.
- [Figure 10.](#) The numeric output for Case A.
- [Figure 11.](#) Case B in graphical form.
- [Figure 12.](#) Case B output in numeric form.
- [Figure 13.](#) The graphic form of Case C.
- [Figure 14.](#) Case C output in numeric form.

Listings

- [Listing 1.](#) Beginning of the program named Fft02?
- [Listing 2.](#) The class named Transform.
- [Listing 3.](#) Performing the transform.
- [Listing 4.](#) The correctAndRecombine method.
- [Listing 5.](#) The remainder of the main method.
- [Listing 6.](#) Case B code.
- [Listing 7.](#) Case C code.

- [Listing 8.](#) The display method.
- [Listing 9.](#) Fft02.java.

General discussion

The purpose of this module is to help you to understand how the Fast Fourier Transform (FFT) algorithm works. In order to understand the FFT, you must first understand the Discrete Fourier Transform (DFT). I explained how the DFT works in an earlier module titled [Fun with Java, How and Why Spectral Analysis Works](#).

There are several different FFT algorithms in common use. In addition, there are many sites on the web where you can find explanations of the mechanics of FFT algorithm. I won't replicate those explanations. Rather, I will explain the underlying concepts that make the FFT possible and illustrate those concepts using a simple program. Hopefully, once you understand the underlying concepts, one or more of the explanations of the mechanics that you find on other sites will make sense to you.

A general-purpose transform

The Fourier transform is most commonly associated with its use in transforming time-domain data into frequency-domain data. However, it is important to understand that there is nothing inherent in the Fourier transform regarding either the time domain or the frequency domain. Rather, the Fourier transform is a general-purpose transform that is used to transform a set of complex data in one domain into a different set of complex data in another domain. It is purely happenstance that it happens to be so valuable in describing the relationship between the time domain and the frequency domain.

Transforming from space domain to wave number domain

For example, my first job after earning a BSEE degree in 1962 was in the Seismic Research Department of Texas Instruments. That is where I had my

first encounter with Digital Signal Processing (*DSP*) . In that job, I did a lot of work with Fourier transforms involving the time domain and the frequency domain. I also did a lot of work with Fourier transforms involving the space domain and the wave-number domain.

Wave number is the name given to the reciprocal of wavelength for compression and shear waves propagating through a medium such as an iron bar, earth, water, or air, and also for electromagnetic waves such as radio and radar propagating through space.

(Those familiar with the subject will know that while compression waves will propagate through water and air, those media won't support shear waves.)

Two-dimensional Fourier transforms

For example, one of the things that we did was to compute two-dimensional Fourier transforms on diagrams representing weighted points in two-dimensional space. We would transform the weighted points in the space domain into points in the wave-number domain.

The weighted points in the space domain represented the locations and amplifications of seismometers in a two-dimensional array on the surface of the earth. Each seismometer was amplified by a different gain factor and polarity. The amplified outputs of the seismometers were added together in various and complex ways intended to enhance signals and suppress noise.

Wave-number response to seismic waves

In this case, the wave number was the reciprocal of the wave length of seismic waves propagating across the array. By plotting the results of the transformation in the wave-number domain, we could estimate which seismic

waves would be enhanced and which seismic waves would be suppressed by the processing being applied to the seismometer outputs.

We could also perform experiments on the computer where we caused the weights to vary with frequency, thus, allowing us to design and place digital filters on the seismometers to optimize the response of the array to earthquake signals while suppressing seismic noise associated with nearby cities and other sources of seismic noise.

A general purpose mathematical transform

I mention all of this simply to illustrate the general nature of the Fourier transform. Once again, the Fourier transform is simply a mathematical process that can be used to transform a set of complex values in one domain into a set of complex values in a different domain.

Before getting into the details of this discussion, I want to refer you to a couple of excellent references on the FFT. Of course, you can find many more by performing a [Google](#) search for the keyword FFT.

Fourier transform images

Many of the images that you will see in this module were produced using an applet named **FftLab** that I originally downloaded from a website named sepwww.stanford.edu/oldsep/hale/FftLab.html. *(As of October 2015, that website no longer exists. However, you can now download the applet from <http://sepwww.stanford.edu/data/media/public/oldsep/hale/FftLab.java>. Also, as of October 2015, you can learn more about the applet's author, Dave Hale [here](#).)*

In order to use the applet to create the illustrations for this document, I changed the name of the applet class to cause it to fit into my file-naming scheme. I also made a couple of minor modifications to the code to force the applet's output to fit into this narrow publication format. Otherwise, I used the applet in its original form. This applet is extremely useful in performing FFT

experiments very quickly and easily. I strongly recommend that you become familiar with it.

As an alternative to downloading the applet from the web page given above, click [here](#) to download a zip file containing the modified source code for the applet along with a Windows batch file that will compile and execute the applet. *((Assuming that you have the Java Development Kit (JDK) and Oracle's **appletviewer** program installed on your computer, you should be able to simply extract the contents of the zip file into an empty folder and double-click on the batch file to run the applet.))*

Will discuss underlying concepts

As mentioned earlier, the FFT algorithm is very complicated. I won't discuss the mechanics of the algorithm in this module. Rather, I will explain the underlying concepts that make the FFT algorithm possible.

Hopefully after reading my explanation of the basic concepts, you will be able to understand the explanation of the mechanics of the algorithm provided by others.

A linear transform

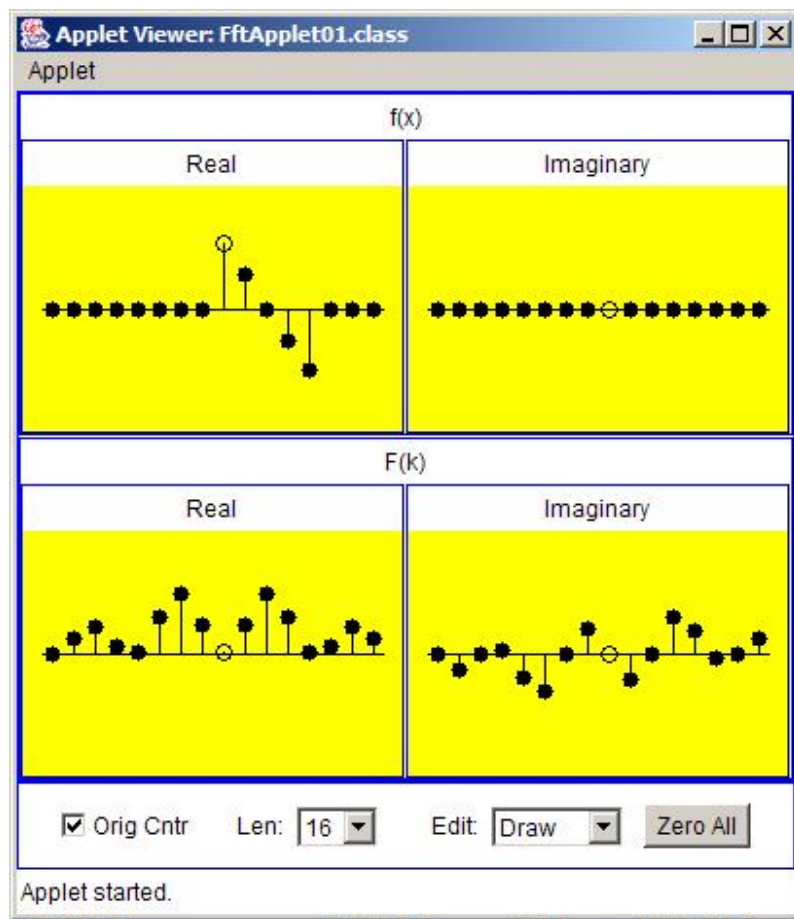
The FFT algorithm is an algorithm that takes advantage of several reasonably well-known facts along with some less well-known facts.

One of those facts is that the Fourier transform is a linear transform. By this, I mean that the transform of the sum of two or more input series is equal to the sum of the transforms of the individual input series. I will attempt to illustrate this in [Figure 1](#), [Figure 2](#), and [Figure 3](#).

Output display of the FFT applet

A sample of the output produced by the FFT applet is shown in [Figure 1](#).

Figure 1. Transform of pulse with negative slope.



An examination of [Figure 1](#) shows that the display produced by the applet contains two sections. One section is labeled $f(x)$ and the other section is labeled $F(k)$.

This is an interactive applet with the ability to transform the complex samples represented by $f(x)$ into complex samples represented by $F(k)$. Alternatively,

the applet can be used to transform complex samples represented by $F(k)$ into complex samples represented by $f(x)$.

Real and imaginary sections

Each section contains two boxes, one labeled Real and the other labeled Imaginary. One box contains a visual representation of a set of real samples and the other box contains a visual representation of a set of imaginary samples.

With one exception, each sample is represented by a black circle. In each box, one of the samples is represented by an empty circle. The empty circle represents an index value of zero. Samples to the right of the sample with the empty circle are samples at positive indices, and samples to the left of the sample with the empty circle are samples at negative indices.

A complex sample

A pair of values, one taken from the Real box and one taken from the Imaginary box, represents a complex sample.

Any of the circles can be interactively moved up or down with the mouse. The value of each sample is represented by the distance of the corresponding circle from the horizontal line.

When a change is made to the value of any sample belonging to either $f(x)$ or $F(k)$, the transformation is recomputed and the display of the other function is modified accordingly. If you modify the value of a sample in $f(x)$, the values in $F(k)$ are automatically modified to show the Fourier transform of $f(x)$. If you modify the value of a sample in $F(k)$, the values in $f(x)$ are automatically modified to show the inverse Fourier transform of $F(k)$.

This is an extremely powerful interactive tool.

Powers of two

Many and perhaps most FFT algorithms require the input series to contain a number of complex samples that is a power of two such as 2, 4, 8, 16, 32, etc. Most FFT algorithms also produce the same number of complex samples in the output as are provided in the input. The FFT algorithm used in this applet is no exception to those rules.

A pull-down list at the bottom of the applet lets the user specify 16, 32, or 64 complex samples for both the input and the output. All of the examples in this module use 16 complex samples for input and output.

Location of the origin

The applet also provides a check box that allows the user to cause the origin (*the empty circle at index value zero*) to either be centered or placed at the left end. The display in [Figure 1](#) has the origin centered. Other displays that I will use later have the origin at the left end.

Other applet controls

The other pull-down list and the button at the bottom of the applet provide other control features that don't need to be discussed here. I strongly urge you to download this applet and experiment with it. The results can be very enlightening.

Back to the concept of the linear transform

Having discussed the features of the interactive FFT tool that I used to produce many of the images in this module, it is time to get back to the discussion of the Fourier transform as a linear transform. The fact that the Fourier transform is a linear transform is illustrated in [Figure 1](#), [Figure 2](#), and [Figure 3](#).

In these three figures, the input series is shown in the real area in the upper left. For simplification, the values of the imaginary part of the input series shown in the upper right are all zero.

Also, for simplification, the zero origin is shown in the center by the value with the empty circle.

The real and imaginary parts of the transform output are shown in the bottom of each figure.

[Figure 1](#) shows an input series consisting of a pulse that starts with a high value at the origin and extends down and to the right for five samples, ending in a large negative value.

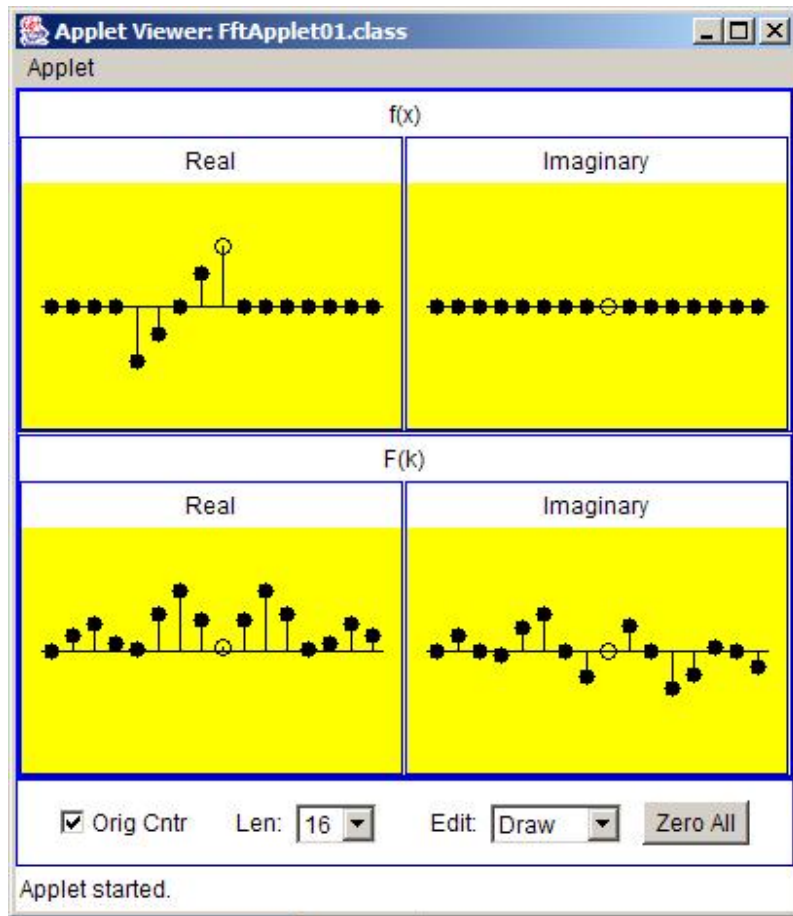
This input series produces a rather complicated transform output series, as can be seen in the bottom two boxes in [Figure 1](#). I will come back to a discussion of the transform output later.

A mirror-image pulse

[Figure 2](#) shown an input series consisting of a pulse that begins with a large negative value four samples to the left of the origin and extends up and to the right ending with a large positive value at the origin. The input series in [Figure 2](#) is the mirror image of the input series in [Figure 1](#) relative to the origin.

Figure 2. Transform of pulse with positive slope.

Figure 2. Transform of pulse with positive slope.



The transform output

Once again, the output from the transform of the input series is shown in the bottom two boxes of [Figure 2](#).

A comparison of the real part of each of the transforms for [Figure 1](#) and [Figure 2](#) shows that the real parts are the same, at least insofar as I was able to control the input by interactively adjusting the locations of the circles using the mouse.

A comparison of the imaginary part of each of the transforms shows that the imaginary parts are the same except for the algebraic sign of each of the values in the imaginary part. The algebraic sign of each of the values in [Figure 2](#) is the reverse of the algebraic sign of each of the values in [Figure 1](#).

Now add the two input series

To demonstrate that the Fourier transform is a linear transform, I will create a new input series that is the sum of the input series from [Figure 1](#) and [Figure 2](#). I will show that the transform of the sum is the sum of the transforms. This is shown in [Figure 3](#).

Figure 3. Transform of the sum of two pulses.

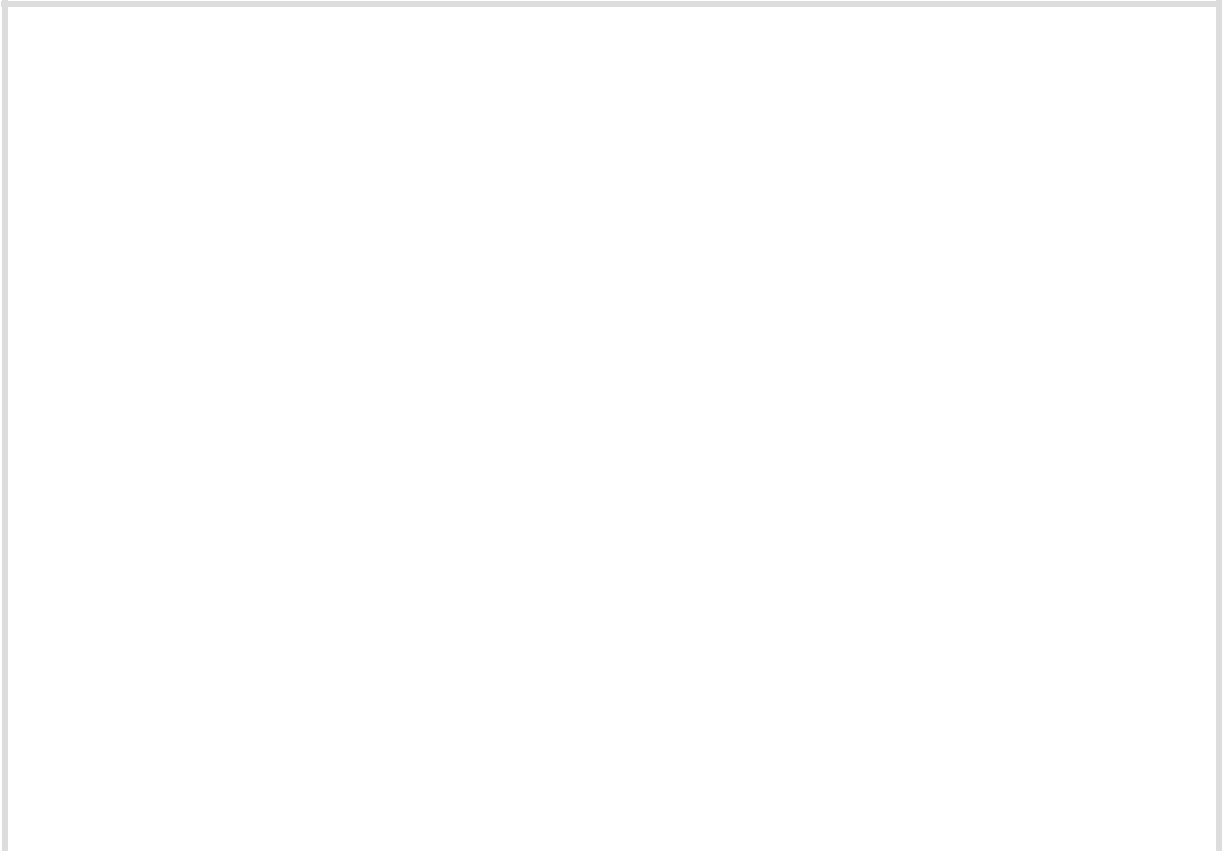
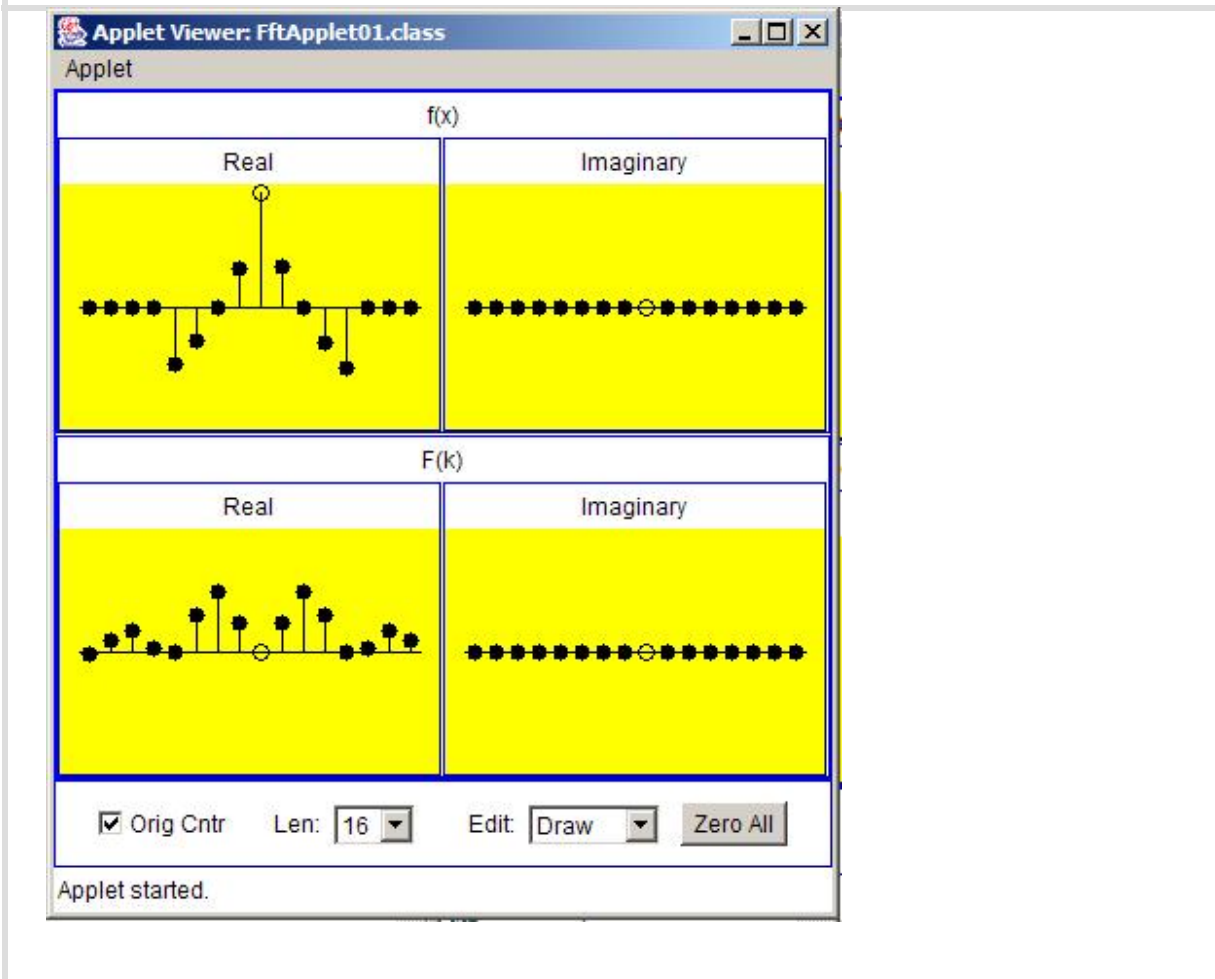


Figure 3. Transform of the sum of two pulses.



The transform of the sum equals the sum of the transforms

[Figure 3](#) shows an input series that is the sum of the individual input series from [Figure 1](#) and [Figure 2](#). This produces a pulse that is symmetric around the origin indicated by the value with the empty circle.

Normalized output

Note that the display of the transform values produced by this applet is normalized so as to keep them in a reasonable range for plotting. As a result, absolute values don't have much meaning. Only relative values have meaning.

The real part is the same

The real part of the transform of the input series in [Figure 3](#) has the same shape as the real parts of the transforms of the input series in [Figure 1](#) and [Figure 2](#). This is what would be produced by adding the real parts of the transforms of the pulses in [Figure 1](#) and [Figure 2](#), and then normalizing the result.

The imaginary part sums to zero

The imaginary part of the transform of the input series in [Figure 3](#) is zero at all sample values. This is what would be produced by adding the imaginary parts of the transforms of the input series in [Figure 1](#) and [Figure 2](#).

(Recall that the values in the imaginary parts of the two earlier transforms had the same magnitude but opposite signs).

Thus, [Figure 1](#), [Figure 2](#), and [Figure 3](#) demonstrate that the transform of the sum of two or more input series is equal to the sum of the transforms of the individual input series. The Fourier transform is a linear transform.

Single sample real pulse (impulse) with a delay

The real part of the transform of a single real sample with a shift relative to the origin has the shape of a cosine curve with a period that is proportional to the reciprocal of the shift. Negative sample values produce cosine curves with negative amplitudes.

A pulse of this type is often referred to an impulse.

The imaginary part of the transform of an impulse with a shift relative to the origin has the shape of a sine curve with a period that is proportional to the reciprocal of the shift. Negative sample values produce sine curves with negative amplitudes.

The magnitude of the transform is the square root of the sum of the squares of the real and imaginary parts at each output sample point. For the case of a single input sample with a shift, that magnitude is constant for all output sample points and is proportional to the absolute value of the sample.

The above facts are illustrated in [Figure 4](#), [Figure 5](#), [Figure 6](#), and [Figure 7](#).

Figure 4. Transform of an impulse with no shift.

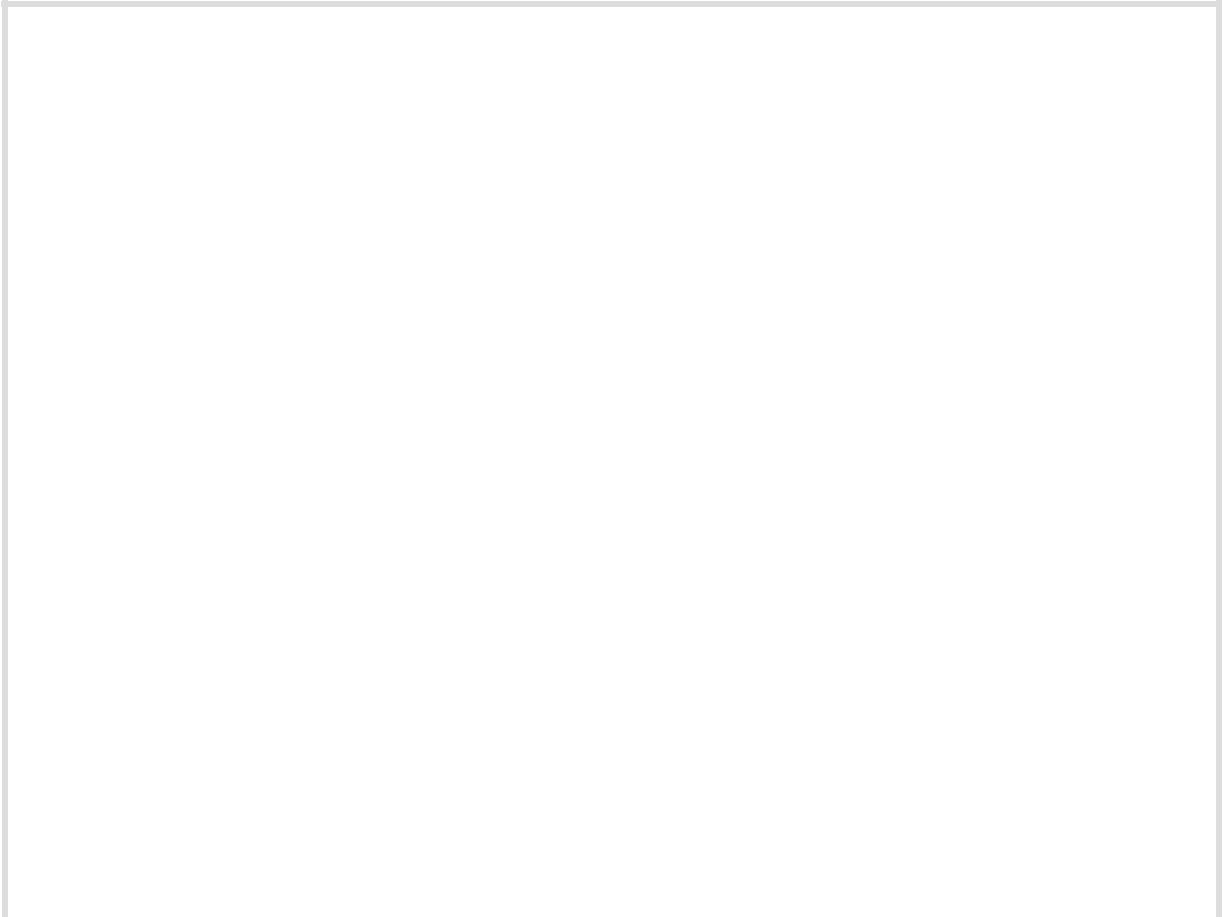
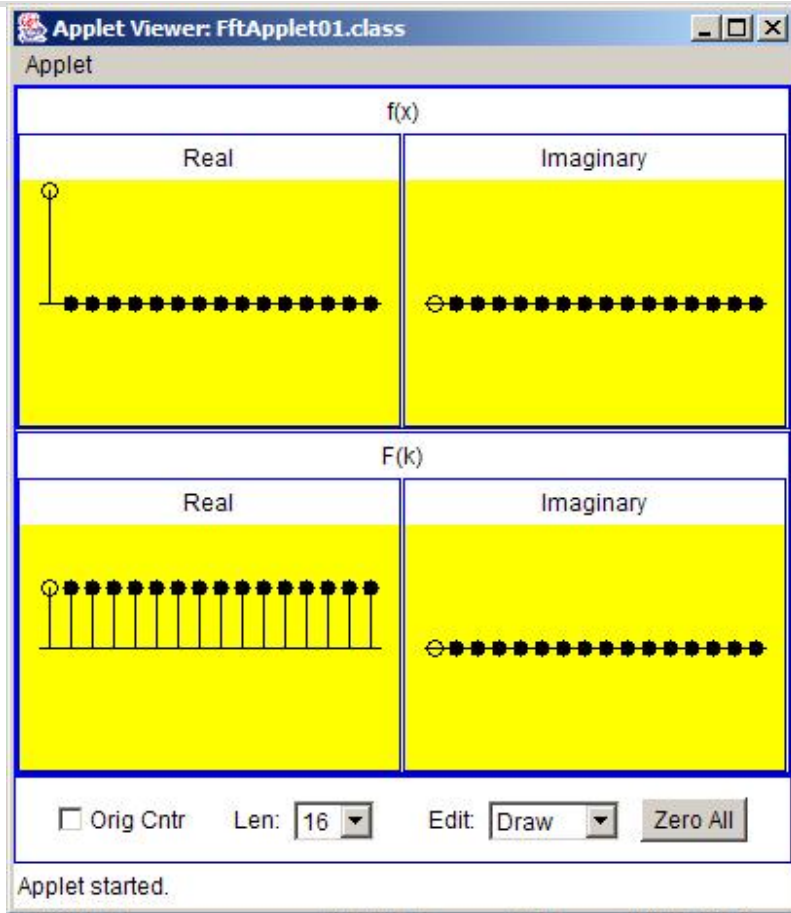


Figure 4. Transform of an impulse with no shift.



A shift of zero

[Figure 4](#) shows the transform of an impulse with a shift of zero relative to the origin.

(Note that in this series of figures, the origin was moved from the center to the left end. Once again, the sample with the empty circle represents the origin.)

Although it isn't obvious, the real part of the transform in [Figure 4](#) has the shape of a cosine curve with a period that is the reciprocal of the shift. Because the shift is zero, the period of the cosine curve is infinite, producing real values that are constant at all output sample values.

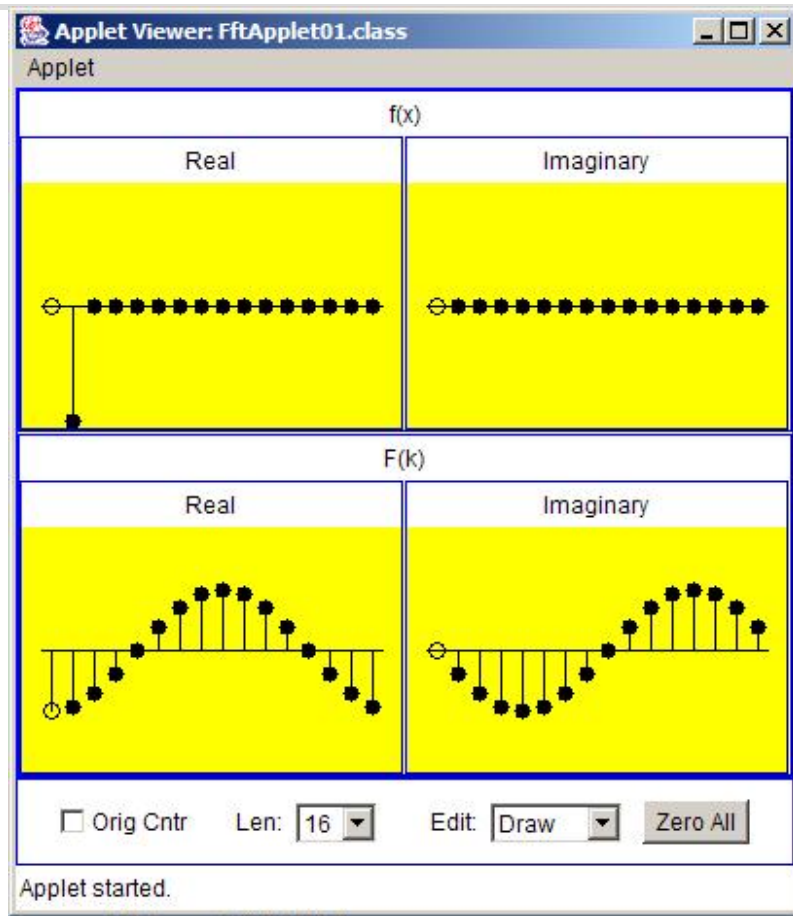
Similarly, the imaginary part of the transform in [Figure 4](#) has a shape that is a sine curve with an infinite period. Thus, it is zero at all output sample values.

A shift of one sample interval

[Figure 5](#) shows the transform of an impulse with a negative value and a shift of one sample interval relative to the origin.

Figure 5. Transform of an impulse with a shift equal to one sample interval and a negative value.

Figure 5. Transform of an impulse with a shift equal to one sample interval and a negative value.



A cosine curve and a sine curve

The shape of the real part of the transform output is an upside down cosine curve. It is upside down because it has a negative amplitude. This is caused by the fact that the input sample has a negative value.

The shape of the imaginary part of the transform is an upside down sine curve.

Number of output samples equals number of input samples

This transform program computes real and imaginary values from zero to an output index that is one output sample interval less than the sampling frequency. The number of output values is equal to the number of samples in the input series. This is very typical of FFT algorithms.

In this case, I set the applet up to accept sixteen input samples and to produce sixteen output samples.

Representing time and frequency

For the moment, let's think in terms of time and frequency. Assume that the input series $f(x)$ is a time series and the output series $F(k)$ is a frequency spectrum.

To make the arithmetic easy, let's assume that the sampling interval for the input time series in the upper left box of [Figure 5](#) is one second. This gives a sampling frequency of one sample per second, and a total elapsed time of sixteen seconds.

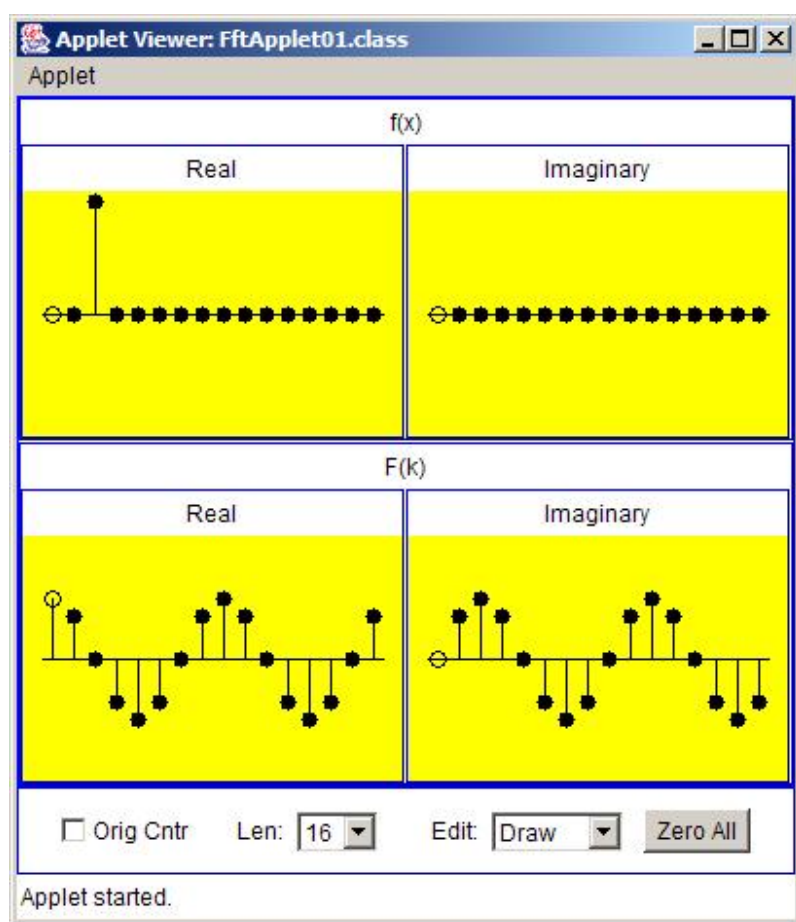
The sine and cosine curves in [Figure 5](#) each go through one complete period between a frequency of zero and the sampling frequency, which is one sample per second. Thus, the period of the sine and cosine curves along the frequency axis is one sample per second. This is the reciprocal of the time shift of one sample interval at a sampling frequency of one sample per second.

Stated differently, the number of periods of the sine and cosine curves in the real and imaginary parts of the transform between a frequency of zero and a frequency equal to the sampling frequency is equal to the shift in sample intervals. A shift of one sample interval produces sine and cosine curves having one period in the frequency range from zero to the sampling frequency. A shift of two sample intervals produces sine and cosine curves having two periods in the frequency range from zero to the sampling frequency, etc. This is illustrated by [Figure 6](#).

A shift of two sample intervals

[Figure 6](#) shows the transform of an impulse with a shift equal to two sample intervals and a positive value.

Figure 6. Transform of an impulse with a shift equal to two sample intervals and a positive value.



The real part of the transform has the shape of a cosine curve with two complete periods between zero and an output index equal to the sampling frequency.

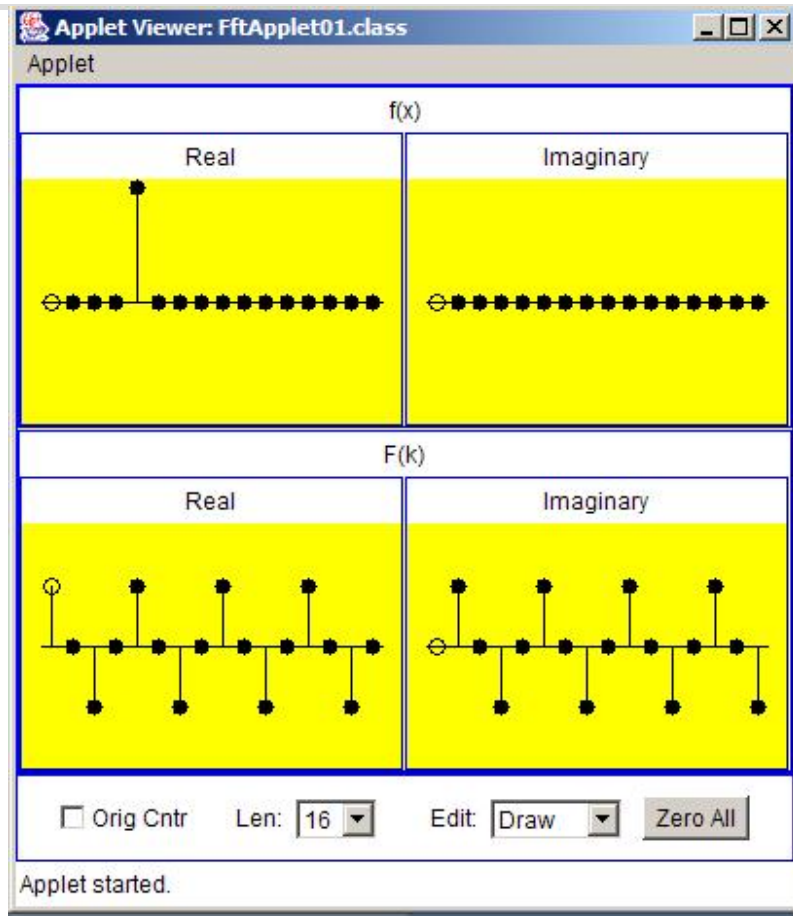
The imaginary part of the transform has the shape of a sine curve with two complete periods within the same output interval. This agrees with the conclusions stated in the previous section.

A shift of four sample intervals

Finally, [Figure 7](#) shows the transform of an impulse with a shift equal to four sample intervals.

Figure 7. Transform of an impulse with a shift equal to four sample intervals and a positive value.

Figure 7. Transform of an impulse with a shift equal to four sample intervals and a positive value.



The cosine and sine curves that represent the real and imaginary parts of the transform each have four complete periods between zero and an output index equal to the sampling frequency.

In this case the cosine and sine curves are very sparsely sampled.

Equations to describe the real and imaginary parts of the transform

The main point is:

If you know the value of a single real sample and you know its position in the series relative to the origin, you can write equations that describe the real and imaginary parts of the transform of that single sample without any requirement to actually perform a Fourier transform.

Those equations are simple sine and cosine equations as a function of the units of the output domain. This is an important concept that contributes greatly to the implementation of the FFT algorithm.

Transformation of a complex series

The FFT algorithm is an algorithm that transforms a series of complex values in one domain into a series of complex values in another domain. The images in the figures discussed so far indicate a transformation of a complex function given by $f(x)$ into another complex function given by $F(k)$. There is nothing in these images to indicate anything about time and frequency.

If the complex part of the input series $f(x)$ is not zero, things get somewhat more complicated. For example, the real and imaginary parts of the transform of an impulse having both real and imaginary parts are not necessarily cosine and sine curves. This is illustrated in [Figure 8](#).

Figure 8. Transform of a complex impulse with a shift equal to two sample intervals.

Applet Viewer: FftApplet01.class

Applet

$f(x)$

Real

Imaginary

$F(k)$

Real

Imaginary

☐ Orig Cntr Len: 16 Edit: Draw Zero All

Applet started.

Although both the real and imaginary parts of the transformed result have the shape of a sinusoid, neither is a cosine curve and neither is a sine curve. Both of the curves are sinusoidal curves that have been shifted along the horizontal output axis moving their peaks and zero crossings away from the origin.

Linearity still applies

Because the Fourier transform is a linear transform, you can transform the real and imaginary parts of the input separately and add the two resulting transforms. The sum of the two transforms represents the transform of the entire input series including both real and imaginary parts. The program that I will discuss later takes advantage of this fact. Once again, the main point is:

Even for a complex input series, if you know the values of the real and imaginary parts of a sample and you know the value of the shift associated with that sample, you can write equations that describe the real part and the imaginary part of the transform results.

Can produce the transform of a time series by the adding transforms of the individual samples

That brings us to the crux of the matter. Given an input series consisting of a set of sequential samples taken at uniform sampling intervals, we know how to write equations for the real and imaginary parts that would be produced by performing a Fourier transform on each of those samples individually.

The input series is the sum of the individual samples

We know that we can consider the input series to consist of the sum of the individual samples, each having a specified value and a different shift. We know that the Fourier transform is a linear transform. Therefore, the Fourier transform of an input series is the sum of the transforms of the individual samples.

If we are clever enough, we can use these facts to develop a computational algorithm that can compute the Fourier transform of a time series much faster than can be obtained using a brute force DFT algorithm. Fortunately, some very clever people have already developed that algorithm. It goes by the name of the Fast Fourier Transform, or FFT algorithm.

Steps in the FFT algorithm

In truth, there are several different forms of the FFT algorithm, and the mechanics of each may be slightly different. At least one, and probably many of the algorithms operate by performing the following steps:

- Decompose an N-point complex series into N individual complex series, each consisting of a single complex sample. The order of the decomposition in an FFT algorithm is rather complicated. It is this order of decomposition, and the order of the subsequent recombination of transform results that causes the FFT algorithm to be so fast. It is also that order that makes the algorithm somewhat difficult to understand. Note that the program that I will discuss later **does not** implement that special order of decomposition and recombination.
- Calculate the transform of each of the N complex series, each consisting of a single complex sample. This treats each complex sample as if it is located at the origin of a complex series. This step is trivial. The real part of the transform of a single complex sample located at the origin of the series is a complex constant whose values are proportional to the real and imaginary values that make up the complex sample. Since the complex input series consists of only one complex sample, there is only one complex value in the complex transform.
- Correct each of the N transform results to reflect the original position of the complex sample in the input series. This involves the application of sine and cosine curves to the real and imaginary parts of the transform. This step is usually combined with the recombination step that follows.
- Recombine the N transform results into a single transform result that represents the transform of the original complex series. This is a very complicated operation in a real FFT algorithm. It must reverse the order of decomposition in the first step described earlier. As mentioned earlier, it is the order of the decomposition and subsequent recombination that minimizes the arithmetic operations required and gives the FFT its tremendous speed. The program that I will discuss later **does not** implement the special order of decomposition and recombination used in an actual FFT algorithm.

A sample program

I want to emphasize at the outset that this program DOES NOT implement an FFT algorithm. Rather, this program illustrates the underlying signal processing concepts that make the FFT possible in a form that is more easily understood than is normally the case with an actual FFT algorithm.

Separate processes in an FFT algorithm

In summary, a typical FFT algorithm performs the following processes:

- Decompose an N-point complex series into N individual complex series, each consisting of a single complex sample.
- Recognize that the complex transform of a single complex sample is equal to the value of the complex sample.
- Correct the transform for each complex sample to reflect the original position of the complex sample in the input series.
- Recombine the N transform results into a single transform result that represents the transform of the original complex series.

This program performs each of the processes listed above. However, it does not perform those processes in the special order used by an FFT algorithm that causes the FFT algorithm to be able to perform those processes at very high speed.

How the processes are implemented

The decomposition process in this program takes the complex samples in the order that they appear in the input complex series. The transform of each complex sample is simply the sample itself. This is the result that would be obtained by actually computing the transform of the complex sample if the sample were the first sample in the series.

The transform result for each complex sample (*the sample itself*) is then corrected for position by applying sine and cosine curves to reflect the actual position of the complex sample within the original complex series.

In order to accomplish the recombination of the corrected transform results, the real and imaginary parts of the corrected transform are added to accumulators. These accumulators are used to accumulate the corrected real and imaginary parts from the corrected transforms for all of the individual complex samples.

Once the real and imaginary parts have been accumulated for all of the complex samples, the real part of the accumulator represents the real part of the transform of the original complex series. The imaginary part of the accumulator represents the imaginary part of the transform of the original complex series. However, an actual transform was never performed on the original complex series.

Three cases are examined

This program creates three separate complex series, applies the processes listed above to each of those series, and displays the results on the screen.

No attempt is made to manage the decomposition and the subsequent recombination in the manner of a true FFT algorithm. Therefore, this program is designed to illustrate the processes involved, and is not designed to provide the speed of a true FFT algorithm.

This program was tested using JDK 1.8 under Windows 7.

Will discuss in fragments

As is my usual approach, I will discuss and explain this program in fragments. A complete listing of the program is provided in [Listing 9](#) near the end of the module.

The program named Fft02

The program begins in [Listing 1](#), which shows the beginning of the controlling class named Fft02 and the beginning of the **main** method.

Listing 1. Beginning of the program named Fft02?

```
class Fft02{

    public static void main(String[] args){
        Transform transform = new
    Transform();
    }
```

Instantiate a Transform object

The first statement in the **main** method instantiates an object of the **Transform** class. This object implements the processes used in an FFT, but **does not** implement those processes in the special order required by an FFT algorithm.

The purpose of an object of the **Transform** class is to illustrate the processes commonly used in an FFT in a manner that is more easily understood than is often the case with an actual FFT algorithm.

I will put the **main** method on the back burner for the moment and explain the class named **Transform** .

The class named Transform

[Listing 2](#) presents the beginning of the class named **Transform** . [Listing 2](#) also presents the beginning of an instance method of that class named **doIt** . The **doIt** method computes and returns the complex transform (*via output parameters*) of an incoming complex series.

Listing 2. The class named Transform.

```
class Transform{  
  
    void doIt(double[] realIn,  
              double[] imagIn,  
              double scale,  
              double[] realOut,  
              double[] imagOut){
```

The method parameters

The **doIt** method receives five incoming parameters. The first two parameters are references to two array objects of type **double** containing the real and imaginary parts of the input series.

The third parameter is a scale factor that is applied to the transform output in an attempt to keep the values in a range suitable for plotting if desired.

The last two parameters are references to array objects of type **double** . The results of performing the transform are used to populate these two arrays. This is the mechanism by which the object returns the transform results to the calling program. It is assumed that all of the elements in these two array objects contain values of zero upon entry to the **doIt** method.

Performing the transform

The body of the **doIt** method is presented in [Listing 3](#). The code in [Listing 3](#) iterates on the input arrays, passing each complex sample contained in those two arrays to a method named **correctAndRecombine** .

Listing 3. Performing the transform.

```
for(int cnt = 0;cnt < realIn.length;cnt++){
    correctAndRecombine(realIn[cnt],
                        imagIn[cnt],
                        cnt,
                        realIn.length,
                        scale,
                        realOut,
                        imagOut);
} //end for loop
} //end doIt
```

The transforms of the complex input samples

Each complex value in the incoming arrays represents both a complex sample and the transform of that complex sample under the assumption that the complex sample appears at the origin of the input series.

Correct for actual position and recombine

The method named **correctAndRecombine** corrects the transform result for each of the complex samples in the series so as to reflect the actual position of the complex sample in the original input series.

Then the method named **correctAndRecombine** adds the corrected transform result into a pair of accumulators, one for the real part and one for the imaginary part. This accomplishes the recombination of the corrected transforms of the input samples in order to produce the transform of the entire original complex input series.

The correctAndRecombine method

The **correctAndRecombine** method is shown in [Listing 4](#). [Listing 4](#) also signals the end of the **Transform** class.

Listing 4. The correctAndRecombine method.

```
void correctAndRecombine(double realSample,
                        double imagSample,
                        int position,
                        int length,
                        double scale,
                        double[] realOut,
                        double[] imagOut){

    //Calculate the complex transform values for
    // each sample in the complex output series.
    for(int cnt = 0; cnt < length; cnt++){
        double angle =

(2.0*Math.PI*cnt/length)*position;

        //Calculate output based on real input
        realOut[cnt] +=

realSample*Math.cos(angle)/scale;
        imagOut[cnt] +=

realSample*Math.sin(angle)/scale;

        //Calculate output based on imag input
        realOut[cnt] -=

imagSample*Math.sin(angle)/scale;
```

Listing 4. The correctAndRecombine method.

```
        imagOut[cnt] +=  
  
        imagSample*Math.cos(angle)/scale;  
    }//end for loop  
    }//end correctAndRecombine  
  
}//end class transform
```

This method accepts an incoming complex sample value and the position in the series associated with that sample. The method corrects the real and imaginary transform values for that complex sample to reflect the specified position in the input series.

After correcting the transform values for the sample on the basis of position, the method updates the corresponding real and imaginary values contained in array objects that are used to accumulate the real and imaginary values for all of the samples.

References to the array objects are received as input parameters. Outgoing results are scaled by an incoming parameter in an attempt to cause the output values to fall within a reasonable range in case someone wants to plot them.

The incoming parameter named length specifies the number of output samples that are to be produced.

Hopefully this explanation will make it possible for you to understand the code in [Listing 4](#).

Note in particular the use of the **Math.cos** and **Math.sin** methods to apply the cosine and sine curves in the correction of the transforms of the individual complex samples. This is used to produce results similar to those shown in [Figure 5](#) through [Figure 7](#).

A real FFT program would probably compute the cosine and sine values only once, put them in a table and extract them from the

table when needed.

Note the use of the **position** and **length** parameters in the computation of the angle that is passed as an argument to the **Math.cos** and **Math.sin** methods.

Also note how the correction is made separately on the real and imaginary parts of the input. This produces results similar to those shown in [Figure 7](#) after those results are added in the accumulators.

Back to the main method

Returning now to the **main** method, the code in [Listing 5](#) prepares the input data and the output arrays for the first case that we will look at. This case is labeled as Case A.

Listing 5. The remainder of the main method.

Listing 5. The remainder of the main method.

```
System.out.println("Case A");
double[] realInA =
{0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
double[] imagInA =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

double[] realOutA = new double[16];
double[] imagOutA = new double[16];

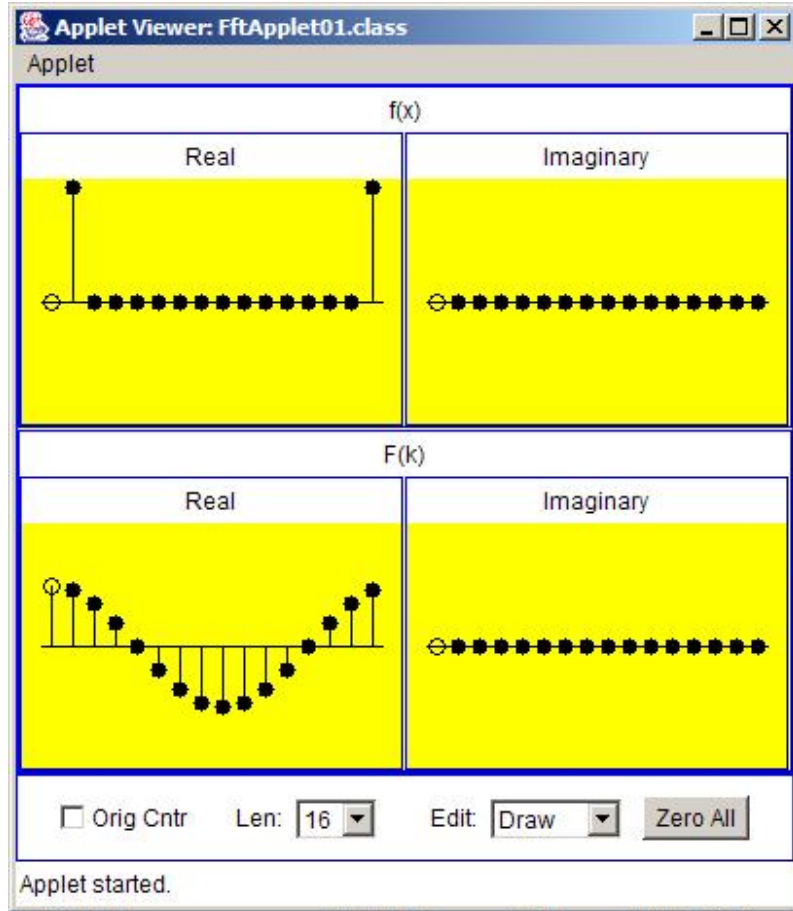
//Perform the transform and display the
// transformed results for the original
// complex series.
transform.doIt(realInA, imagInA, 2.0, realOutA,
imagOutA);
display(realOutA, imagOutA);
```

Note that for Case A, the input complex series contains non-zero values only in the real part. Also, most of the values in the real part are zero.

The graphic form of Case A

Case A is shown in graphic form in [Figure 9](#). As you can see, the input series consists of two non-zero values in the real part. All the values in the imaginary part are zero.

Figure 9. Case A. Transform of a real sample with two non-zero values.



The real part of the transform of the complex input series looks like one cycle of a cosine curve. All of the values in the imaginary part of the transform result are zero.

The numeric output for Case A

As you saw in [Listing 5](#), the code in the main method calls a method named **display** to display the complex transform output in numeric form on the screen. The output produced by [Listing 5](#) is shown in [Figure 10](#). (Note that I

manually inserted line breaks to force the material to fit in this narrow publication format.)

Figure 10. The numeric output for Case A.

```
Case A
Real:
1.0 0.923 0.707 0.382 0.0 -0.382 -0.707 -0.923
-1.0 -0.923 -0.707 -0.382 0.0 0.382 0.707 0.923
imag:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

If you plot the real and imaginary values in [Figure 10](#), you will see that they match the transform output shown in graphic form in [Figure 9](#).

Case B code

The code from the **main** method for Case B is shown in [Listing 6](#). Note that the input complex series contains non-zero values in both the real and imaginary parts.

Listing 6. Case B code.

Listing 6. Case B code.

```
System.out.println("\nCase B");
double[] realInB =
{0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
double[] imagInB =
{0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,-1};

double[] realOutB = new double[16];
double[] imagOutB = new double[16];

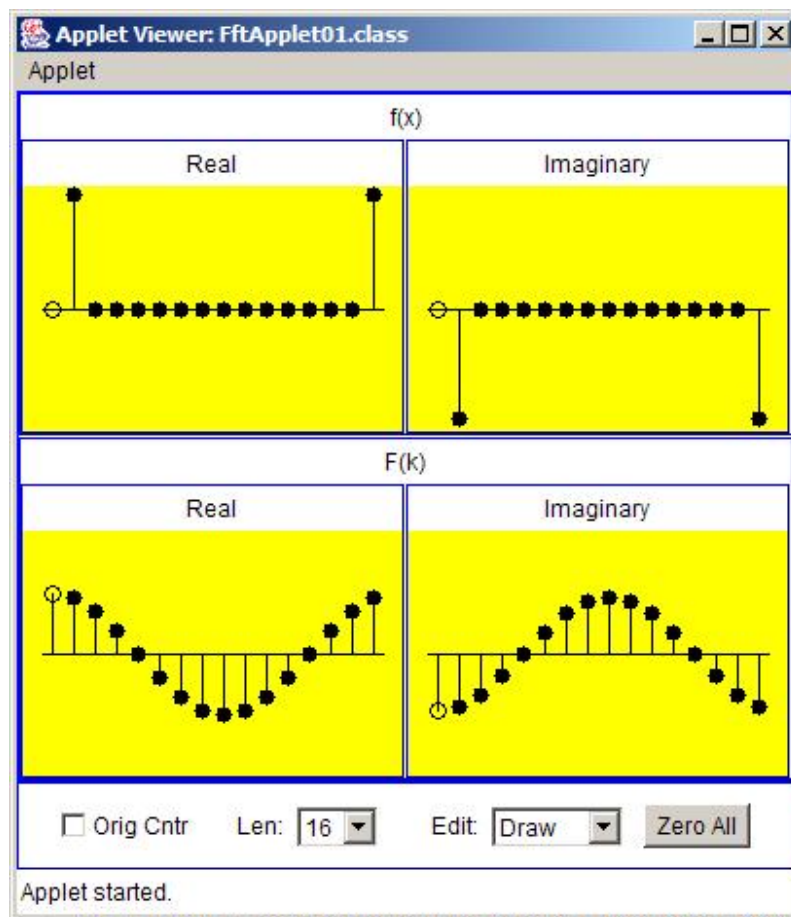
transform.doIt(realInB,imagInB,2.0,realOutB,
imagOutB);
display(realOutB,imagOutB);
```

Case B in graphical form

Case B is shown in graphical form in [Figure 11](#).

Figure 11. Case B in graphical form.

Figure 11. Case B in graphical form.



Case B output in numeric form

The output from the code in [Listing 6](#) is shown in [Figure 12](#).

Figure 12. Case B output in numeric form.

Figure 12. Case B output in numeric form.

Case B

Real:

```
1.0 0.923 0.707 0.382 0.0 -0.382 -0.707 -0.923
-0.999 -0.923 -0.707 -0.382 0.0 0.382 0.707
0.923
```

imag:

```
-1.0 -0.923 -0.707 -0.382 0.0 0.382 0.707 0.923
1.0 0.923 0.707 0.382 0.0 -0.382 -0.707 -0.923
```

If you plot the values for the real and imaginary parts from [Figure 12](#), you will see that they match the real and imaginary output shown in [Figure 11](#).

Case C code

The code extracted from the **main** method for Case C is shown in [Listing 7](#).

Listing 7. Case C code.

Listing 7. Case C code.

```
System.out.println("\nCase C");
double[] realInC =
    {1.0,0.923,0.707,0.382,0.0,-0.382,-0.707,
     -0.923,-1.0,-0.923,-0.707,-0.382,0.0,
     0.382,0.707,0.923};
double[] imagInC =
    {0.0,-0.382,-0.707,-0.923,-1.0,-0.923,
     -0.707,-0.382,0.0,0.382,0.707,0.923,
     1.0,0.923,0.707,0.382};

double[] realOutC = new double[16];
double[] imagOutC = new double[16];

transform.doIt(realInC,imagInC,16.0,realOutC,
imagOutC);
display(realOutC,imagOutC);
```

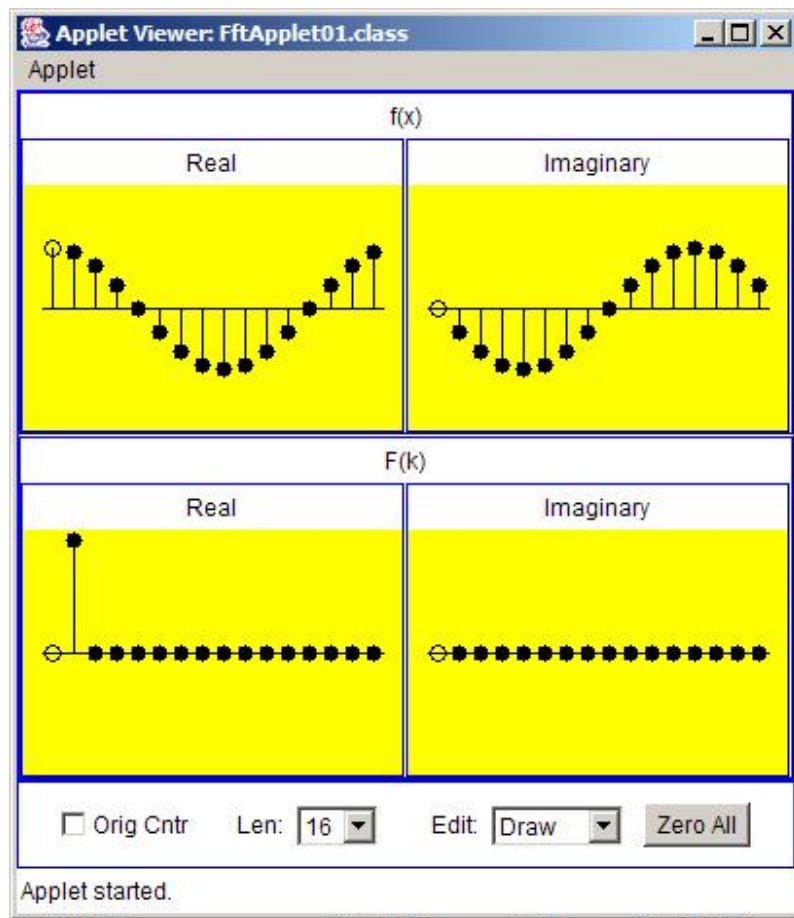
The complex input series for Case C is a little more complicated than that for either of the previous two cases. Note in particular that the input complex series contains non-zero values in both the real and imaginary parts. In addition, very few of the values in the complex series have a value of zero.

(The values of the complex samples actually describe a cosine curve and a negative sine curve as shown in [Figure 13](#).)

The graphic form of Case C

Case C is shown in graphic form in [Figure 13](#).

Figure 13. The graphic form of Case C.



The Fourier transform is reversible

One of the interesting things to note about [Figure 13](#) is the similarity of [Figure 13](#) and [Figure 5](#). These two figures illustrate the reversible nature of the Fourier transform.

If I had used a positive input real value instead of a negative input real value in [Figure 5](#), the input of [Figure 5](#) would look exactly like the output in [Figure 13](#), and the output of [Figure 5](#) would look exactly like the input of [Figure 13](#).

With that as a hint, you should now be able to figure out how I used a mouse and drew the perfect sine and cosine curves in [Figure 13](#). In fact, I didn't draw them at all. Rather, I used my mouse and drew the output, and the applet gave me the corresponding input automatically.

Case C output in numeric form

The output produced by the code in [Listing 7](#) is shown in [Figure 14](#).

Figure 14. Case C output in numeric form.

```
Case C
Real:
0.0 0.999 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
imag:
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

If you plot the real and imaginary input values from [Listing 7](#), you will see that they match the input values in [Figure 13](#). If you plot the real and imaginary output values in [Figure 14](#), you will see that they match the output values shown in [Figure 13](#).

[Listing 7](#) signals the end of the **main** method.

The display method

[Listing 8](#) shows the code for a simple method named **display** . The purpose of the **display** method is to display a real series and an imaginary series, each contained in an incoming array object of type **double** . The **double** values are truncated to no more than four digits before displaying them. Then they are displayed on a single line.

Listing 8. The display method.

```
static void display(double[] real,
                   double[] imag){
    System.out.println("Real: ");
    for(int cnt=0;cnt < real.length;cnt++){
        System.out.print(((int)(1000.0*real[cnt]))
                        /1000.0 + " ");
    }//end for loop
    System.out.println();

    System.out.println("imag: ");
    for(int cnt=0;cnt < imag.length;cnt++){
        System.out.print(((int)(1000.0*imag[cnt]))
                        /1000.0 + " ");
    }//end for loop
    System.out.println();
}//end display

}//end class Fft02
```

[Listing 8](#) also signals the end of the controlling class named **Fft02** .

Run the program

I encourage you to copy and compile the program that you will find in [Listing 9](#). Experiment with different complex input series.

I also encourage you to download the applet from <http://sepwww.stanford.edu/data/media/public/oldsep/hale/FftLab.java> or from [here](#) and experiment with it as well. Compare the numeric output produced by this program with the graphic output produced by the applet.

Finally, I encourage you to examine the source code for the applet. Concentrate on that portion of the source code that performs the FFT. Hopefully, what you have learned in this module will make it easier for you to understand the source code for the FFT.

Summary

In this module, I have explained some of the underlying signal processing concepts that make the FFT possible. I illustrated those concepts in a program designed specifically to be as simple as possible while still illustrating the concepts.

Now that you understand those concepts, you should be able to better understand explanations of the mechanics of the FFT algorithm that appear on various websites.

Complete program listings

A complete listing of the program is provided in [Listing 9](#) below.

Listing 9. Fft02.java.

Listing 9. Fft02.java.

```
/*File Fft02.java Copyright 2004, R.G.Baldwin  
Rev 4/30/04
```

This program DOES NOT implement an FFT algorithm. Rather, this program illustrates the underlying FFT concepts in a form that is much more easily understood than is normally the case with an actual FFT algorithm. The steps in the implementation of a typical FFT algorithm are as follows:

1. Decompose an N-point complex series into N individual complex values, each consisting of a single complex sample. The order of the decomposition in an FFT algorithm is rather complicated. It is this order of decomposition, and the order of the subsequent recombination of transform results that causes the FFT to be so fast. It is also that order that makes the algorithm somewhat difficult to understand. This program does not implement that order of decomposition and recombination.
2. Calculate the transform of each of the N complex samples, treating each as if it were located at the beginning of the complex series. This step is trivial. The real part of the transform of a single complex sample located at the beginning of the series is a complex constant whose values are proportional to the real and imaginary values that make up the complex sample.
3. Correct each of the N transform results to reflect the actual position of the complex sample in the series. This involves the application of sine and cosine curves to the real and imaginary

Listing 9. Fft02.java.

parts of the transform. This step is usually combined with the recombination step that follows.

4. Recombine the N transform results into a single transform result that represents the transform of the original complex series. This is a very complicated operation in a real FFT algorithm. It must reverse the order of decomposition in the first step described earlier. As mentioned earlier, it is the order of the decomposition and subsequent recombination that minimizes the arithmetic operations required and gives the FFT its tremendous speed. This program does not implement the special order of decomposition and recombination used in an actual FFT algorithm.

This program creates three separate complex series, applies the processes listed above to each of those series, and displays the results on the screen. No attempt is made to manage the decomposition and the subsequent recombination in the manner of a true FFT algorithm. Therefore, this program is designed to illustrate the processes involved, and is not designed to provide the speed of a true FFT algorithm.

The decomposition process in this program takes the complex samples in the order that they appear in the input complex series.

The transform of each complex sample is simply the sample itself. This is the result that would be obtained by actually computing the transform of the complex sample if the sample were the

Listing 9. Fft02.java.

first sample in the series.

The transform result for each complex sample is then corrected by applying sine and cosine curves to reflect the actual position of the complex sample within the complex series.

The real and imaginary parts of the corrected transform results are then added to accumulators that are used to accumulate the corrected real and imaginary parts from the corrected transforms for all of the individual complex samples.

Once the real and imaginary parts have been accumulated for all of the complex samples, the real part of the accumulator represents the real part of the transform of the original complex series. The imaginary part of the accumulator represents the imaginary part of the transform of the original complex series.

Tested using SDK 1.4.2 under WinXP

*****/

```
class Fft02{

    public static void main(String[] args){

        //Instantiate an object that will implement
        // the processes used in an FFT, but not in
        // the order required by an FFT algorithm.
        Transform transform = new Transform();

        //Prepare the input data and the output
```

Listing 9. Fft02.java.

```
// arrays for Case A. Note that for this
// case, the input complex series contains
// non-zero values only in the real part.
// Also, most of the values in the real part
// are zero.
System.out.println("Case A");
double[] realInA =
    {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
double[] imagInA =
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

double[] realOutA = new double[16];
double[] imagOutA = new double[16];

//Perform the transform and display the
// transformed results for the original
// complex series.
transform.doIt(realInA, imagInA, 2.0, realOutA,
               imagOutA);
display(realOutA, imagOutA);

//Process and display the results for Case B.
// Note that the input complex series
// contains non-zero values in both the real
// and imaginary parts. However, most of the
// values in the real and imaginary parts are
// zero.
System.out.println("\nCase B");
double[] realInB =
    {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
double[] imagInB =
    {0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,-1};

double[] realOutB = new double[16];
double[] imagOutB = new double[16];
```

Listing 9. Fft02.java.

```
transform.doIt(realInB, imagInB, 2.0, realOutB,
               imagOutB);
display(realOutB, imagOutB);

//Process and display the results for Case C.
// Note that the input complex series
// contains non-zero values in both the real
// and imaginary parts. In addition, very
// few of the values in the complex series
// have a value of zero. (The values of the
// complex samples actually describe a cosine
// curve and a sine curve.)
System.out.println("\nCase C");
double[] realInC =
    {1.0, 0.923, 0.707, 0.382, 0.0, -0.382, -0.707,
     -0.923, -1.0, -0.923, -0.707, -0.382, 0.0,
     0.382, 0.707, 0.923};
double[] imagInC =
    {0.0, -0.382, -0.707, -0.923, -1.0, -0.923,
     -0.707, -0.382, 0.0, 0.382, 0.707, 0.923,
     1.0, 0.923, 0.707, 0.382};

double[] realOutC = new double[16];
double[] imagOutC = new double[16];

transform.doIt(realInC, imagInC, 16.0, realOutC,
               imagOutC);
display(realOutC, imagOutC);

} //end main
//=====//

//The purpose of this method is to display
// a real series and an imaginary series,
```

Listing 9. Fft02.java.

```
// each contained in an incoming array object
// of type double. The double values are
// truncated to no more than four digits
// before displaying them. Then they are
// displayed on a single line.
static void display(double[] real,
                    double[] imag){
    System.out.println("Real: ");
    for(int cnt=0;cnt < real.length;cnt++){
        System.out.print(((int)(1000.0*real[cnt]))
                          /1000.0 + " ");
    }//end for loop
    System.out.println();

    System.out.println("imag: ");
    for(int cnt=0;cnt < imag.length;cnt++){
        System.out.print(((int)(1000.0*imag[cnt]))
                          /1000.0 + " ");
    }//end for loop
    System.out.println();
} //end display

} //end class Fft02
//=====//

//This class applies the processes normally used
// in an FFT algorithm. However, this class does
// not apply those processes in the special order
// required of an FFT algorithm. It is that
// special order that minimizes the arithmetic
// requirements of an FFT algorithm and causes it
// to be very fast. The purpose of an object of
// this class is to illustrate the processes in a
// more easily understood fashion that is often
// the case with an actual FFT algorithm.
class Transform{
```


Listing 9. Fft02.java.

```
void doIt(double[] realIn,double[] imagIn,
         double scale,double[] realOut,
         double[] imagOut){
    //Each complex value in the incoming arrays
    // represents both a complex sample and the
    // transform of that complex sample under the
    // assumption that the complex sample appears
    // at the beginning of the series.
    //Correct the transform result for each of
    // the complex samples in the series to
    // reflect the actual position of the complex
    // sample in the series. Add the corrected
    // transform result into accumulators in
    // order to produce the transform of the
    // original complex series.
    for(int cnt = 0;cnt < realIn.length;cnt++){
        correctAndRecombine(realIn[cnt],
                           imagIn[cnt],
                           cnt,
                           realIn.length,
                           scale,
                           realOut,
                           imagOut);
    }//end for loop
}//end doIt

//=====//

//This method accepts an incoming complex
// sample value and the position in the series
// associated with that sample. The method
// calculates the real and imaginary transform
// values associated with that complex sample
// when it is located at the specified
// position. Then it updates the corresponding
```

Listing 9. Fft02.java.

```
// real and imaginary values contained in array
// objects used to accumulate the real and
// imaginary values for all of the samples.
// References to the array objects are received
// as input parameters. Outgoing results are
// scaled by an incoming parameter in an
// attempt to cause the output values to fall
// within a reasonable range in case someone
// wants to plot them.
void correctAndRecombine(
    double realSample, double imagSample,
    int position, int length, double scale,
    double[] realOut, double[] imagOut){
    //Calculate the complex transform values for
    // each sample in the complex output series.
    for(int cnt = 0; cnt < length; cnt++){
        double angle =
            (2.0*Math.PI*cnt/length)*position;
        //Calculate output based on real input
        realOut[cnt] +=
            realSample*Math.cos(angle)/scale;
        imagOut[cnt] +=
            realSample*Math.sin(angle)/scale;

        //Calculate output based on imag input
        realOut[cnt] -=
            imagSample*Math.sin(angle)/scale;
        imagOut[cnt] +=
            imagSample*Math.cos(angle)/scale;
    } //end for loop
} //end correctAndRecombine

} //end class transform
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1486-Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm
- File: Java1486.htm
- Published: 01/04/05

Baldwin explains the underlying signal processing concepts that make the Fast Fourier Transform (FFT) algorithm possible.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1490-2D Fourier Transforms using Java, Part 1

Learn how the space domain and the wavenumber domain in two-dimensional analysis are analogous to the time domain and the frequency domain in one-dimensional analysis. Learn about some practical examples showing how 2D Fourier transforms and wavenumber spectra can be useful in solving engineering problems involving antenna arrays.

Revised: Tue Oct 20 11:48:13 CDT 2015

This page is included in the following book: [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Two separate programs](#)
 - [Digital signal processing.\(DSP\)](#)
 - [Will use in subsequent modules](#)
 - [Viewing tip](#)
 - [Figures](#)
- [General discussion](#)
 - [Time domain and frequency domain](#)
 - [A one-dimensional Fourier transform](#)
 - [Time domain data is purely real](#)
 - [The space domain](#)
 - [Time and space are analogous](#)
 - [Frequency and wavenumber are analogous](#)
 - [Period and wavelength are analogous](#)

- Some real world examples
 - A commercial radio station
 - Basic concepts
 - A one-dimensional space
 - How can we compute the array response?
 - Three example wavenumber spectra
 - Plots of 3D surfaces
 - A wavenumber filter
 - Let's apply some weights
 - A weighted three-element array
 - A three-element array with negative weighting
 - What can we learn from these scenarios?
- Extending into two dimensions
 - Move the array to a table top
 - The 2D wavenumber response of a linear array
- A two-dimensional array
 - Computing the wavenumber response
- Arrays are used in various applications
 - Radio astronomy
 - Seismology
 - Sonar
 - Radar
 - Petroleum exploration
 - Image processing
- Summary
- What's next?
- Miscellaneous

Preface

This is the first module of a two-part series. In this module, I will:

- Explain the conceptual and computational aspects of 2D Fourier transforms
- Explain the relationship between the space domain and the [wavenumber](#) domain
- Provide sufficient background information that you will be able to appreciate the importance of the 2D Fourier transform

Two separate programs

In [Part 2](#) of this series, I will present and explain two separate programs. One program consists of a single class named **ImgMod30**. The purpose of this class is to satisfy the computational requirements for forward and inverse 2D Fourier transforms. This class also provides a method for rearranging the spectral data into a more useful format for plotting. The second program named **ImgMod31** will be used to test the 2D Fourier transform class, and also to illustrate the use of 2D Fourier transforms for some well known sample surfaces.

A third class named **ImgMod29** will be used to display various 3D surfaces resulting from the application of the 2D Fourier transform. I explained this class in an earlier module titled [Plotting 3D Surfaces using Java](#)..

Digital signal processing (DSP)

This and the following module will cover some technically difficult material in the general area of Digital Signal Processing, or DSP for short. As usual, the better prepared you are, the more likely you are to understand the material. For example, it would be well for you to already understand the one-dimensional Fourier transform before tackling the 2D Fourier transform. If you don't already have that knowledge, you can learn about one-dimensional Fourier transforms by studying the following **modules** :

- [1478 Fun with Java, How and Why Spectral Analysis Works](#)

- [1482 Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#)
- [1483 Spectrum Analysis using Java, Frequency Resolution versus Data Length](#)
- [1484 Spectrum Analysis using Java, Complex Spectrum and Phase Angle](#)
- [1485 Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain](#)
- [1486 Fun with Java, Understanding the Fast Fourier Transform \(FFT\) Algorithm](#)

Will use in subsequent modules

The 2D Fourier transform has many uses. I will use the 2D Fourier transform in several future modules involving such diverse topics as:

- Processing image pixels in the wavenumber domain
- Advanced steganography (*hiding messages in images*)
- Hiding watermarks and trademarks in images

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

- [Figure 1.](#) A standing wave on a wire.
- [Figure 2.](#) Three example wavenumber spectra.
- [Figure 3.](#) A three-element array with weighted sensors.
- [Figure 4.](#) The 2D wavenumber response of a linear array.
- [Figure 5.](#) Wavenumber response of a two-dimensional array.

- [Figure 6.](#) Image processing in the space domain.

General discussion

Time domain and frequency domain

In my earlier modules on DSP, you learned about the relationship between the time domain and the frequency domain. For example, you learned that time has only one dimension. In the real world, time only goes forward.

(In the computer world, we can make it appear that time can also go backwards, but this still constitutes only one dimension.)

The important point is that time can only go forward or backwards. It cannot go sideways.

A one-dimensional Fourier transform

You learned that you can perform a one-dimensional Fourier transform to transform your data from the time domain into the frequency domain. Similarly, you can perform an inverse one-dimensional Fourier transform to transform your data from the frequency domain back into the time domain.

You learned about several characteristics of Fourier transforms. For example, you learned that a Fourier transform is both linear and reversible. You either have learned or you will learn in a future module that convolution in the time domain is equivalent to multiplication in the frequency domain, and that convolution in the frequency domain is equivalent to multiplication in the time domain.

You learned that with enough computational power, you can easily transform a given set of data back and forth between these two domains.

This makes it possible to use the domain of your choice to perform a given signal processing operation, even if the results need to be delivered in the other domain.

Time domain data is purely real

Although it is possible to use the Fourier transform to transform a set of complex data from one domain to another domain, real-world time domain data is not complex data. Rather, it is purely real. Assuming that the data in one domain is always purely real leads to some simplification of the computational requirements for performing the Fourier transform. In general, most of the previous DSP modules assumed real data in the time domain and complex data in the frequency domain.

The space domain

In this module, we will extend the concept of the Fourier transform from the time domain into the space domain. In making this extension, we will encounter some significant additional complexity. For example, while time is one-dimensional, space is three-dimensional. While you can only move forward and backwards in time, you can move up, down, forward, backward, and from side to side in space.

(In order to keep the complexity of this module in check, we will assume that space is only two-dimensional, allowing movement up, down, and from side to side only. This will serve us well later for such tasks as image processing. Three-dimensional Fourier transforms are beyond the scope of this module.)

It is also possible and very common to combine time domain signal processing with space domain signal processing. However, that also is

beyond the scope of this module.

Time and space are analogous

We will consider the space domain to be analogous to the time domain, with the stipulation that the space domain has two dimensions. The unit of measure in the time domain is usually seconds, or some derivative thereof. The unit of measure in space is usually meters, or some derivative thereof.

As with the time domain, we will assume that all space domain surfaces are purely real (*as opposed to being complex*) . This will allow us to simplify our computations when performing the 2D Fourier transform to transform our data from the space domain into the wavenumber domain.

(I will point out that from a practical viewpoint this assumption is much more limiting in the space domain than in the time domain. Complex space domain functions are quite common in such areas as antenna array processing.)

Frequency and wavenumber are analogous

We will consider the wavenumber domain to be analogous to the frequency domain. The unit of measure in the frequency domain is cycle per second, or some derivative thereof. The unit of measure in the wavenumber domain is cycles per meter or some derivative thereof.

Period and wavelength are analogous

The reciprocal of the typical unit of measure in the frequency domain is seconds per cycle, commonly referred to as the period. The reciprocal of the

typical unit of measure in the wavenumber domain is meters per cycle, commonly referred to as the wavelength.

Some real world examples

With all of this as background, I will begin by discussing some real world engineering problems for which the solution lies in an understanding of the wavenumber domain. I will use these examples to show some of the practical uses of 2D Fourier transforms.

Following that (in [Part 2 of this series](#)) , I will present and explain a class that you can copy and use to perform 2D Fourier transforms. Then I will present and explain a program that exercises and tests the 2D Fourier transform class for some common 3D surfaces.

A commercial radio station

Assume that you have just acquired an FCC license to build and operate a new commercial radio station in a small town in west Texas. As is frequently the case in west Texas, your town is situated at the intersection of two highways. One highway runs northeast and southwest. The other highway runs northwest and southwest. The two highways are generally perpendicular to one another. Like many highways in west Texas, each of these highways is straight as an arrow with very few curves.

Where people live

Your town has a small business district at the intersection of the two highways. Beyond that, most of the people who live in your town (*and who will listen to your radio station*) live along the two highways. Thus most of the population lives in the directions of northeast, southwest, northwest, and southeast from the center of town. There are very few people living in the

directions of north, south, east, and west. That real estate is mostly populated by cows and cotton fields.

A limit on the transmitting power

Your new FCC license places a limit on the amount of power that you will be allowed to transmit. You would like to use that available power to reach a many human listeners as possible. If you simply construct an omnidirectional transmitting antenna and start broadcasting, approximately half of the power that you transmit will be available mostly to cows and cotton plants. As a result, the amount of power, and hence the reach of the power that you transmit to human listeners will be less than you would like for it to be.

A directional transmitting antenna

While the FCC won't allow you to increase the amount of power that you transmit, they will allow you to control the directions in which you choose to transmit that power. You will probably hire an expert in the transmission of radio signals to design a directional transmitting antenna system, which will broadcast most of the available power in the directions where the people live. Ideally, the antenna system will transmit very little of the available power in the directions of the cows and the cotton fields.

The design of the antenna system

The antenna designer will have many tools at her disposal. Whether or not she uses wavenumber terminology, many of the calculations that she performs will depend on wavenumber concepts. She will be concerned about the reciprocal wavenumber (*wavelength*) of the radio signals that will be broadcast. She will be concerned with the lengths of the active elements in the antenna system, and the distance between active elements if she chooses to use an array of active elements.

Basic concepts

I will leave the radio station scenario at this point and discuss some more basic concepts. I will return to the radio station scenario later. We will need to start with simpler things and work our way up to the radio station scenario.

Much of this discussion will be couched in terms of receiving signals rather than transmitting signals. (*For most people, receiving is easier to understand than transmitting.*) However, most of the conclusions that we reach regarding antenna systems used for receiving signals are also applicable to antenna systems used for transmitting signals.

A one-dimensional space

Just to get us started down the right path, we will temporarily constrain space to have only one dimension. We will discuss the propagation of waves along a taut wire, as well as the measurement of the waves propagating along that wire.

Assume that a wire is fastened at both ends, is fairly taut, and is suspended between two walls so that it is free to move up and down only. Assume that we attach two sensors to the wire, one meter apart, and that each of these sensors is capable of generating an electrical signal that is proportional to the vertical displacement of the wire at the point where the sensor is attached. If the wire goes up, the sensor generates a positive signal. If the wire goes down, the sensor generates a negative signal.

Standing waves

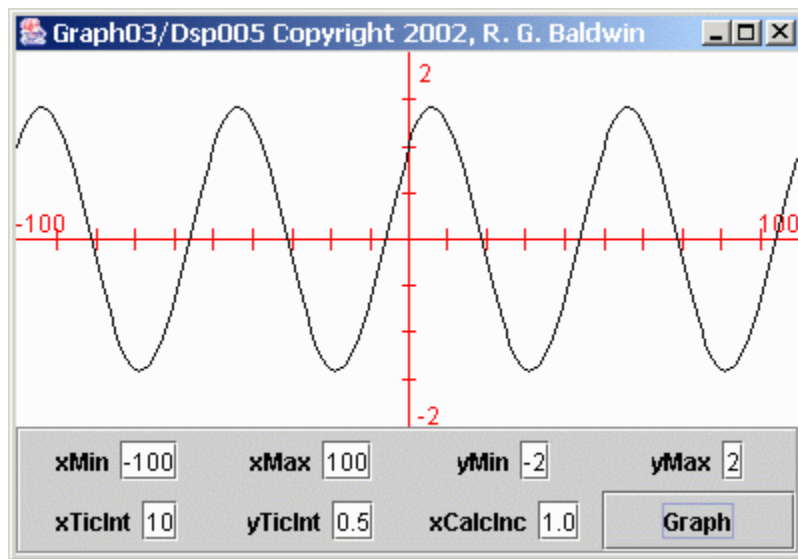
There are two ways that we can approach this analysis. If the wire is very long, we can think in terms of a deformation pulse that propagates along the wire passing by our sensors once and only once. This would fall into the category of *transient analysis*.

On the other hand, if the wire is shorter, we can think in terms of vibrating one end of the wire in such a way that a standing wave will develop on the wire. This is probably the easier of the two approaches to understand because you may have created standing waves on a rope as a child.

What does a standing wave look like?

Once a standing wave is set up on the wire, it will take on an appearance very similar to the sine wave shown in [Figure 1](#).

Figure 1. A standing wave on a wire.



[Figure 1](#) represents a snapshot taken at a single instant in time. Obviously the wire doesn't remain in the position shown in [Figure 1](#) for very long. Rather a single point on the wire will move up and down with time with the overall appearance being as shown in [Figure 1](#). The distance between any

two positive peaks is what we would refer to as the wavelength of the standing wave.

Add the sensor output signals

Now consider what would happen if we were to electronically add the electrical outputs produced by the two sensors. The result would depend on the distance between the sensors relative to the wavelength of the standing wave. For example, if the two sensors were exactly one wavelength apart, the two sensors would move up and down in unison, and the sum of the two signals would be double the signal level produced by either sensor alone. The result would be the same as if the two sensors were located at the same place on the wire.

A one-half wavelength spacing

On the other hand, if the two sensors were exactly one-half wavelength apart, one would be going up when the other is going down, and the electrical output from one would cancel the electrical output from the other. In space processing terminology, this would be referred to as a *null point*.

Change the wavelength of the standing wave

Now consider what would happen if you were to leave the two sensors in the same locations as before and do something to the wire to change the wavelength of the standing wave. The output from the sum of the two sensors would range from zero to maximum as the wavelength relative to the sensor separation varies from one-half wavelength to one full wavelength. For a one-half wavelength separation, the output would be zero. For a full wavelength separation, the output would be at its maximum.

A two-element array

We could refer to our two sensors as a *two-element array* , and we could refer to the output produced by the sum of the two sensors as the response of the array. We could plot the response versus wavelength. However, in the same sense that it is more common to plot the response of electrical filters versus frequency (*instead of period*) , it is probably most common to plot the response of arrays versus wavenumber (*instead of wavelength*) . Therefore, in this module, we will plot the response of the array versus wavenumber.

How can we compute the array response?

This is where Fourier transforms come into play. We could compute the response of our one-dimensional array for waves propagating along the length of the wire by treating the elements of the array as samples in space and performing a one-dimensional Fourier transform on the elements of the array.

A sampled space series

In this case, we would consider the array elements to constitute samples taken in space in the same way that we consider a sampled time series to constitute samples taken in time. In other words, the array elements constitute a sampled space series. The Fourier transform of a sampled time series is the frequency spectrum of the time series. The Fourier transform of a sampled space series is the wavenumber spectrum of the space series.

Three example wavenumber spectra

To set the stage for what we will be seeing later, [Figure 2](#) depicts three different two-element arrays with different spacing in the three (*black and white*) images across the top of the figure. The wavenumber response for each of the three arrays is shown in the three images in the bottom row of images in [Figure 2](#).

(In [Figure 2](#), the array elements are represented by the white dots on the black background.)

Figure 2. Three example wavenumber spectra.



Plots of 3D surfaces

The images shown in [Figure 2](#) were produced using the class named **ImgMod29**, which I explained earlier in the module titled [Plotting 3D Surfaces using Java](#). You can refer back to that module for a detailed explanation of the display format. Briefly, each of the six individual images in [Figure 2](#) is a plot of a 3D surface, with the elevation of the surface at any particular point being indicated by the color at that point based on the colors on the calibration scale below each plot.

(The calibration scale is the strip that changes color in a smooth gradient from black through blue, cyan, green, yellow, red, and

white with black at the left end and white at the right end.

The three images in the top row of [Figure 2](#) with the white dots on the black background represent the array elements for each of the three arrays. The three images in the bottom row of [Figure 2](#) represent the wavenumber response of the corresponding arrays in the top row.)

Black, white, and the colors in between

The lowest (*algebraic*) elevation in the plot is colored black. Hence the backgrounds are black in the top three plots. The highest elevation is colored white. The array elements are white in the top three plots.

Between black and white, the elevation is given by the color scale below the plot with the lowest elevation on the left of the scale and the highest elevation on the right. Thus, a green elevation is about half way between the lowest and highest elevations. Blue elevations are near the low end. Red elevations are near the high end. Cyan and yellow elevations fall in between as shown by the calibration scale.

The maximum array output

All three wavenumber plots have a maximum response at the center, which is the zero wavenumber origin. In effect, this corresponds to infinite wavelength. If the wavelength is infinite, it doesn't matter what the separation between the elements is, they will all move up and down in unison and their electrical outputs will add constructively to produce a maximum output.

Response of the leftmost array

Now consider the leftmost pair of images where the two array elements are relatively close together. The wavenumber response of this array has a pair of null points about midway between the center and either end, as indicated by the blue and black colors. This is the wavenumber for which the element spacing is an exact multiple of one-half of the wavelength, causing the elements to move in equal but opposite directions in response to the wave motion. Thus, the output from one element cancels the output from the other element producing zero voltage in the sum.

This same pair of images shows high responses at either end as indicated by the red areas. This is the wavenumber for which the element spacing is an exact multiple of one wavelength.

The Nyquist folding wavenumber

If this were a frequency spectrum analysis, we would say that the end points are at the Nyquist folding frequency, which is one-half of the sampling frequency. Thus, we can say that in the wavenumber domain, the end points on the two leftmost plots are at the Nyquist folding wavenumber, which is one-half the sampling wavenumber.

As in the frequency domain, the wavenumber spectrum is periodic. The section of the wavenumber spectrum that we are viewing represents one complete period of a periodic wavenumber spectrum ranging from minus the folding wavenumber on the left, through zero wavenumber at the center, to plus the folding wavenumber on the right.

(If you are already familiar with this sort of thing, you may have figured out that the separation between our elements in this example is twice the sampling distance that determines the location of the folding wavenumber.)

Estimating the array response from the colors

The leftmost array response shows a white area at the center.

(The white area is split by the vertical component of red axes drawn on the plot. The color of the axes has nothing to do with elevations on the surface. I simply decided to draw them in red to make them stand out.)

This array response also shows the two black areas mentioned earlier. You can use the orange, yellow, green, and cyan locations on the calibration scale to estimate the response of the array to different wavenumber values between maximum and minimum.

Additional separation between array elements

Now consider the two images in the center of [Figure 2](#) where the wavenumber range is the same as the wavenumber plot on the left. The elements for this array are separated more than the elements in the leftmost pair of images. Again, the wavenumber response for this array has a maximum value at the origin, which is at the center of the wavenumber response in the lower image. In addition, this array response has a high (*red*) response at four other wavenumber zones (*for a total of five*), whereas the array on the left had only three red zones. Similarly, this array has a low response (black and blue) at four different wavenumber zones whereas the array on the left has a low response at only two wavenumber zones.

Thus, changing the separation between the elements has a significant impact on the wavenumber response of the array.

Separate the elements even more

Finally, consider the pair of images on the right where the elements are even further apart. This array response has even more peaks and valleys than the

other two.

A wavenumber filter

The sum of the outputs from an array of sensor elements represents a form of wavenumber filter (*much as the correct combination of resistors, capacitors, and inductors represents a frequency filter*) . If we need to pass signals having one wavenumber and to suppress signals having a different wavenumber, we may be able to adjust the separation between the elements so as to put a peak on the desirable wavenumber and to put a null point on the undesirable wavenumbers.

A two-element array is fairly limiting

Of course, with only two elements, we don't have very many degrees of freedom to work with. We could exercise more control over our wavenumber filter if we had more elements. We could do even better if we had the ability to give each element a different weight (*including a negative weight*) when the signals from all the elements are added together. Finally, we could do even better still if we had the ability to insert a programmable time delay (*phase shift*) into the output from each of the elements before adding them together.

(The use of programmable time delays falls in the category of a space series that is complex rather than being purely real. Thus, that topic is beyond the scope of this module.)

Let's apply some weights

Now let's modify our scenario and see what we can learn in the process. We are going to increase the size of the array from two to three elements. We

are also going to assume that we can apply amplification, sign reversal, or both to the element output signals before adding those signals together. The results are shown in [Figure 3](#). We will compare the results in [Figure 3](#) with the results discussed earlier in [Figure 2](#), so this may be a good time for you to open another copy of this module in a separate browser window if you haven't already done so.

Figure 3. A three-element array with weighted sensors.



The leftmost pair of images

The leftmost pair of images in [Figure 3](#) is similar to the leftmost pair of images in [Figure 2](#), except that we added a third element in [Figure 3](#).

All three elements in [Figure 3](#) are weighted equally prior to summation. The separation between the left and center elements is the same as in [Figure 2](#). The separation between the center and right elements is the same as the separation between the left and center elements.

The wavenumber response

The most noticeable thing about the wavenumber response for this three-element array is that the central peak is narrower than the central peak for the two-element array at the left of [Figure 2](#). In addition, the trough between the central peak and the peaks at the ends is deeper, broader, and probably flatter (*although the degree of flatness is hard to determine from this plotting format*) .

Could continue adding elements to the array

Although I won't demonstrate it, I can tell you that if I were to continue adding elements in this manner to increase the length of the array, the central peak and the peaks at the folding wavenumbers would continue getting narrower, and the trough between the peaks would continue getting deeper and probably flatter.

A more selective wavenumber filter

In other words, when viewed as a wavenumber filter, a long array with more elements is a more selective wavenumber filter than a short array with fewer elements. By properly designing an array to act as a wavenumber filter, it is possible to cause that filter to be very selective.

When we use a properly designed array to produce a directional antenna, it is possible to produce a highly directional antenna (*and avoid wasting our valuable radio frequency energy by sending it to cows and cotton plants*).

A weighted three-element array

Continuing with our three-element array scenario, let's take a look at the center pair of images in [Figure 3](#) and compare them with the leftmost images in [Figure 3](#).

For this case, the array still contains three elements with the same spacing as before. However, the electrical output from the center element is amplified to make it twice as strong as the outputs from the other two elements before the three electrical signals are added together.

Note that the array elements are no longer shown as being white against a black background in the very top of the top-center image. Instead, the center element is white but the other two elements are greenish indicating different weights.

It is a little hard to tell what this does to the central peak in the wavenumber response, but it definitely changes the shape of the response in the trough between the peaks. Whether or not this would be a beneficial change would depend on the problem being addressed.

A three-element array with negative weighting

Finally, take a look at the rightmost pair of images in [Figure 3](#). Once again, the array contains three elements and the center element is weighted twice as heavily as the other two. In addition, the sign of the electrical signals from the two outer elements is inverted before the three are added together.

Note that the black color represents the smallest algebraic value. Negative values are smaller than positive values. Therefore, the background (which represents a weight of zero) has changed from black to something between green and blue. The two elements with the negative weights are shown as black and the element with the positive weight is shown as white.

This has a major impact on the wavenumber response of the three-element array.

There is no longer a peak at a wavenumber value of zero. Rather, there is now a null point at zero wavenumber as indicated by the black and blue colors at the center of the plot. There is now a peak on each side of zero (*as indicated by the white and red colors*) , half way between zero and the folding wavenumber.

Two full peaks but at different locations

If you consider the peaks at the ends of the wavenumber response for the leftmost and center images in [Figure 3](#) to each represent only half a peak (*with the other half being off the scale to the left and the right*) , all three scenarios have two complete peaks in their wavenumber responses.

(You could think in terms of printing the wavenumber response on a piece of paper, cutting it out, and taping the two ends together to form a continuous ring. As you made a complete traversal of the ring, you would encounter two peaks.)

However, the locations of the two peaks for the rightmost array are at completely different wavenumber values than are the peaks for the other two arrays. The two peaks exhibited by the rightmost array are in the locations of the two nulls for the center array. Similarly, the null points for the rightmost array are in the same locations as the two peaks for the center array.

What can we learn from these scenarios?

We learn that we can have a significant impact on the wavenumber response of an array by increasing the number of elements in the array. We can also

have a significant impact on the wavenumber response by applying weights, *(including sign changes)* , to the electrical signals produced by the array elements before adding them together.

Extending into two dimensions

Now let's complicate things a bit by extending our array analysis into two dimensions. Up to this point, we have assumed that our sensors were attached to a wire that was free to move up and down only. As such, waves impinging on the array were constrained to approach the array from one end or the other. In this case the wavenumber was completely determined by the wavelength of the wave.

(For our purposes, the wavelength is given by the ratio of propagation speed in meters per second to frequency in cycles per second. Canceling out the units leaves us with wavelength in meters per cycle.)

Move the array to a table top

Let's move our array of sensors from the wire to a large sheet of metal on the top of a table. For the time being, we will still place the elements in a line with uniform spacing. However, we will now assume that a wave can impinge on the array from any direction along the surface of the sheet of metal.

(For simplicity, we will assume that there is some sort of insulation between the sheet of metal and the table top to prevent waves from impinging on the array from below.)

What does a wave look like in this scenario?

Imagine a piece of corrugated sheet metal or fiber glass. (*Material like this is sometimes used to build a roof on a patio.*) When you look at it from one end, it looks something like the sine wave in [Figure 1](#). However, if you keep it at eye level and slowly turn it in the horizontal plane, the distance between the peaks will appear to become shorter and shorter until finally you don't see any peaks at all. What you see at that point is something that appears to have the same thickness from one end to the other. This is the view that one of our sensors sees as the wavefront of an impinging wave.

The angle of attack is important

We now have a much more complex situation. If the waves continue to impinge on the array from one end or the other, the situation will be exactly the same as when the sensors were on the wire. However, the *apparent* wavenumber or wavelength of a wave as seen by the array will depend on the angle of attack.

(There is now a difference between the actual wavelength or wavenumber and the apparent wavelength or wavenumber as seen by the array.)

Infinite wavelength

For example, if the wave impinges on the array from a broadside direction, all of the sensors will move up and down in unison regardless of their separation and regardless of the actual wavelength of the wave. For this case, the wave will appear to the array to have infinite wavelength or zero wavenumber.

(A linear array has no ability to filter on the basis of wavenumber for waves that impinge on the array from the broadside direction. All waves from that direction appear to have zero wavenumber. This will lead us later to consider the use of a two-dimensional array.)

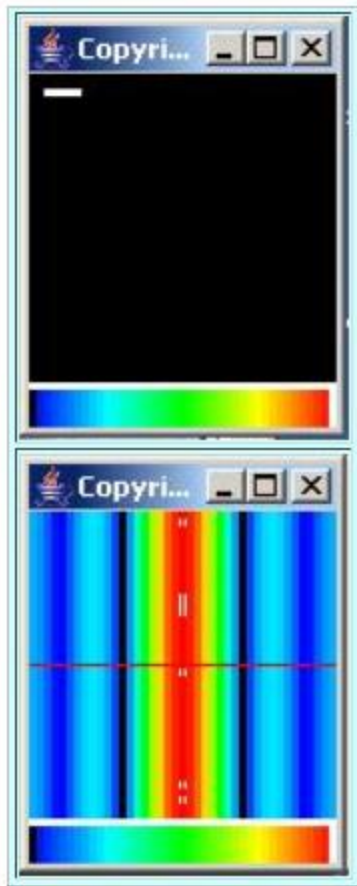
The 2D wavenumber response of a linear array

[Figure 4](#) shows the two-dimensional wavenumber response for a five element linear array with equal weighting for all of the elements. The array is shown at the top. The wavenumber response of the array is shown at the bottom.

(In this case, I placed all five elements on adjacent points on the space sampling grid with no spaces in between. This places them so close together that you can't visually separate them in the image and they appear as a white line in a black background.)

Figure 4. The 2D wavenumber response of a linear array.

Figure 4. The 2D wavenumber response of a linear array.



The response for a constant wavenumber

If you were to draw a circle centered on the crosshairs (*axes*) in the center of the wavenumber response, the points on that circle would represent a fixed wavenumber for a wave arriving from any direction. The value of the response at any particular point on the circle would indicate the response of the array to a wave having that wavenumber from that direction.

Response versus direction

If the diameter of the circle is larger than the width of the red vertical band, and if you were to plot that response versus direction, you would see that the response is maximum for the two directions that are broadside to the array and the response tends to drop off as the direction approaches the end-fire direction of the array.

Symmetry

You would also notice quite a lot of symmetry. For example, the maximum response occurs in two directions that are 180 degrees apart. In fact, if you pick any direction and a given wavenumber, the response is the same for that direction and for the direction that is 180 degrees around from that direction.

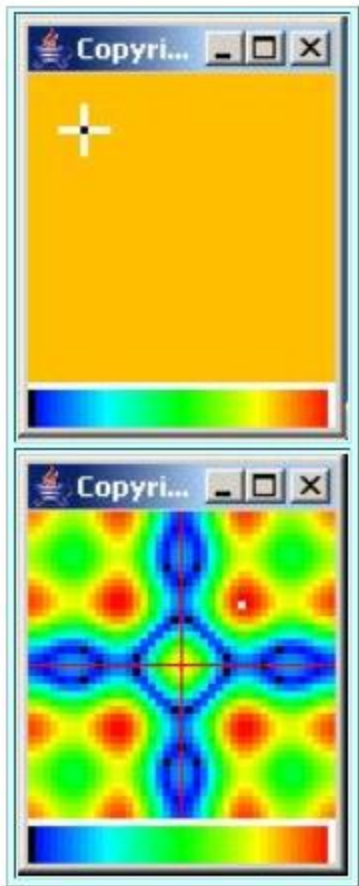
Not good for the radio transmitter

This wouldn't be a very good design for the radio station that I described earlier. If one of the broadside directions of the array faces northeast and the other faces southwest, then the people who live in the northwest and southeast directions wouldn't receive a very good signal from your transmitter. You need a design that maximizes the power in the four directions where the people live, and that minimizes the power in the other directions. To accomplish that, we will need a two-dimensional array in place of our one-dimensional linear array.

A two-dimensional array

We will achieve the desired array response by using an array having thirteen elements in the form of a cross with very specific weighting applied to each element prior to summation. The weighted array and the wavenumber response of the array are shown in [Figure 5](#).

Figure 5. Wavenumber response of a two-dimensional array.



Computing the wavenumber response

The wavenumber response of the array (*shown at the bottom in [Figure 5](#)*) was produced by performing a 2D Fourier transform on the weighted array (*shown at the top in [Figure 5](#)*).

The center element in the array was weighted by -4.5 before summation. (*The signal from this element was amplified by a factor of 4.5 and the sign of the signal was inverted prior to summation.*)

The twelve remaining elements were weighted by a factor of 1.0 before summation.

A constant wave number

Regardless of direction, the wavelength or wavenumber of the RF energy transmitted by a commercial radio station is the same and it doesn't change over time, *(unless the frequency on which the station broadcasts changes)* .

Once again, we can determine the wavenumber response of the array for a given wavenumber from any direction by drawing a circle on the response plot, centered at the origin, with the radius of the circle equal to the specific wavenumber of interest.

Assume a wavenumber

Assume that the wavenumber of interest in our case is exactly equal to the distance from the origin to the white spot in the upper right quadrant of the wavenumber response plot. This white spot represents the maximum response of the array.

(If our computation had perfect accuracy, there would be a white spot at the same location in all four quadrants.)

Draw a circle centered on the origin having a radius that causes the circle to go through the white spot.

Determine the response versus direction

The color corresponding to the response at any point on the circle represents the response of the array to that wavenumber for waves arriving

from that direction (*or in the case of a transmitter, for waves being transmitted in that direction*) .

The circle passes through yellow, red, and white (*indicating a high response*) in the general directions of northeast, southeast, southwest, and northwest. This means that a strong RF signal will be transmitted to the people living along the highways in those four directions.

The circle passes through green, cyan, and blue (*indicating a lower response*) in the general directions of north, south, east, and west. This means that very little of the precious RF energy will be transmitted to the cotton plants and the cows that live in those directions.

A possible solution to the problem

Thus, an array of active transmitter elements arranged and weighted as shown at the top of [Figure 5](#) might be a reasonable design solution for your radio station. (*However, I suspect that an experienced RF engineer would have a much more sophisticated solution.*)

In any event, you have now seen one possible practical example of the use of a 2D Fourier transform.

Arrays are used in various applications

Although this example was admittedly somewhat contrived, it is not far fetched. Arrays similar to those that I have been discussing are widely used in the technology area of spatial signal processing.

Radio astronomy

Perhaps the application that is most familiar to the general public (*due to widespread publicity and a very popular movie*) is the [Paul Allen radio telescope](#) used in the [Search for Extraterrestrial Intelligence \(SETI\)](#).

In the past, much of this work has been done using a very large dish antenna known as the [Arecibo Radio Telescope](#) in Puerto Rico. Efforts are now underway involving an alternative approach that uses a large [array](#) of small dishes instead of one large dish.

By properly processing and then summing the outputs produced by the dishes in the array, the users will be able to steer the telescope and possibly to also eliminate strong sources of interference.

Seismology

Arrays of seismometers are used by U.S. government agencies to monitor for seismic signals produced by earthquakes in locations nearly halfway around the earth.

By applying complex, frequency dependent weighting factors to the seismometer outputs before summing them, the arrays can be tuned to provide a complex response in wavenumber space. For example, the arrays can be processed to form response beams looking in different directions with a beam width that is relatively constant across a wide band of interesting frequencies. In addition, null points in the wavenumber response can be created to suppress seismic noise that originates from specific points on the earth such as mines, rock quarries, and cities.

The design and analysis of such array systems use 2D (*and sometimes 3D*) Fourier transforms. Because the weights that are applied are produced by complex frequency filters, the transform programs that are used must treat both the space domain data and the wavenumber data as complex (*instead of being purely real as in the examples in this module*) .

Sonar

Probably ninety percent of all sonar systems currently installed on surface ships and submarines use arrays for steering and processing both active and passive sonar. In almost all cases, these are 3D arrays. Some of the arrays

contain multiple sensors on the surface of a portion of a sphere. Some contain multiple sensors located along slats that are mounted on a frame much like the staves on a barrel. Some are located on the sides of the vessel. There are probably numerous other geometries in use as well.

A Fourier transform program used with these arrays would normally have to be a 3D Fourier transform program capable of transforming from complex space functions to complex wavenumber functions.

Radar

One of the reasons that sonar is typically processed using arrays has to do with the wavelength of the signals and the operating environment. It is usually not practical to physically move a sonar sensor large enough to do the job in order to cause it to look in different directions. Thus arrays of small sensors are used with the ability to steer beams electronically in order to look in different directions.

Because of the shorter wavelengths involved, typical radar sensors are usually small enough that they can be physically turned and tilted. Thus, it is not unusual to see radar sensors turning around and tilting up and down. Although I'm not personally aware of any applications that use arrays of radar sensors, my suspicion is that there probably are some being used in fixed air surveillance operations.

Petroleum exploration

A large percentage of petroleum exploration involves the insertion of a powerful surge of acoustic energy into the ground (*or into the ocean*) and listening for and recording the echo signals returned by the various layers of the earth. By moving across the earth and repeating this process, a [profile](#) of the earth's layering can be produced. An experienced exploration geophysicist can examine the profiles and reach conclusions as to the likelihood that a particular stratum contains petroleum.

Exploration geophysicists have been using arrays of sensors for this purpose for at least the past 55 years according to my personal knowledge, and probably for many years before that.

Image processing

Image processing in the wavenumber domain

While the examples described above are interesting, they are beyond the scope of anything that I can demonstrate online. However, there are several interesting applications using 2D Fourier transforms that I can demonstrate online. One of those applications is image processing.

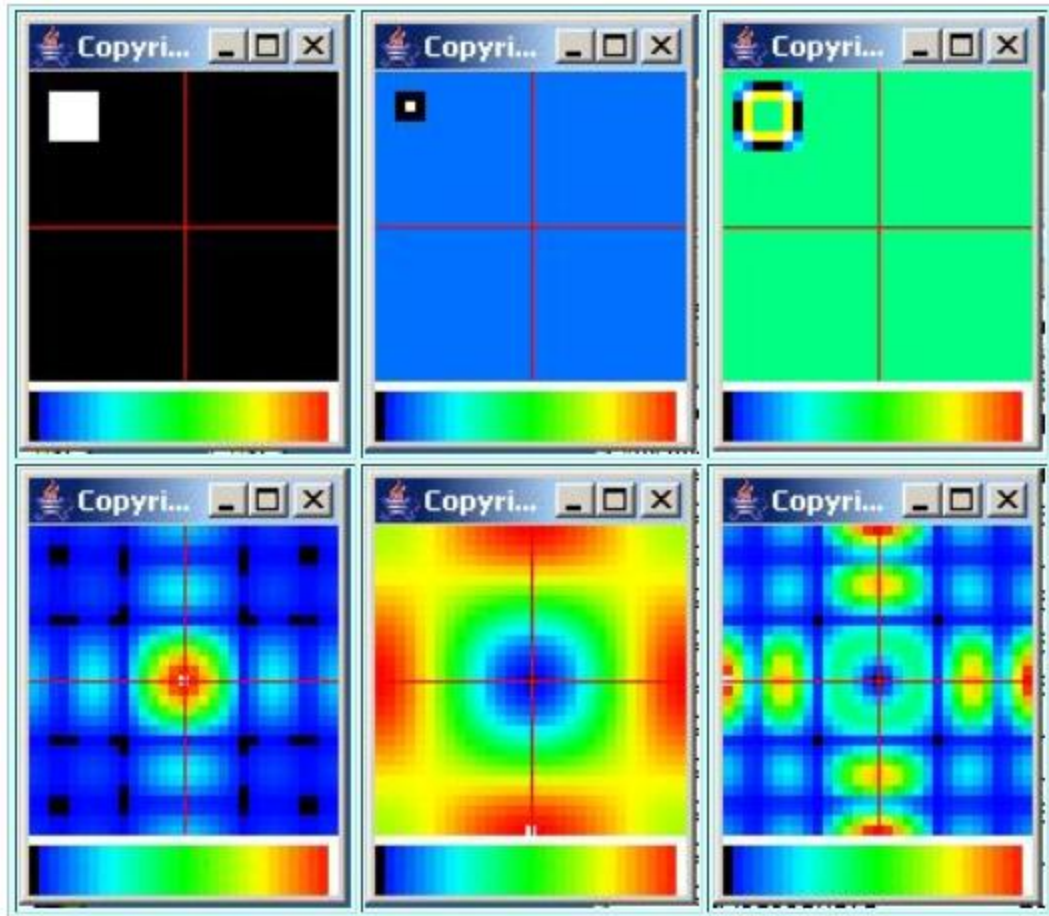
Future modules will show how to use 2D Fourier transforms for such purposes as softening images, sharpening images, doing edge detection on images, etc. For this application, it is satisfactory to use a 2D Fourier transform program that assumes that the space domain data is purely real. Therefore, the program that I will present and explain in [Part 2](#) of this module will make that assumption.

Image processing in the space domain

The 2D Fourier transform will be used in future modules to help explain how and why 2D image convolution behaves the way it does. A preview of that material is shown in [Figure 6](#).

Figure 6. Image processing in the space domain.

Figure 6. Image processing in the space domain.



Convolution versus multiplication

Convolution in the space domain is equivalent to multiplication in the wavenumber domain, and vice versa.

A simple space function and its wavenumber spectrum

The top left image in [Figure 6](#) shows a simple 3D surface in space consisting of a raised square. The wavenumber spectrum of that surface is

shown in the lower left image in [Figure 6](#). Note that the spectrum has a peak at a wavenumber value of zero with low values at the higher wave numbers near the edges. The peak in the center is relatively narrow with respect to the folding wave number at the edges.

A 2D convolution operator and its spectral response

The image in the upper center of [Figure 6](#) shows a typical 2D convolution operator consisting of a value of +8 in the center surrounded by eight coefficients each having a value of -1.

The wavenumber spectral response of that convolution operator is shown in the lower center. Note that it has peaks at the folding wavenumbers on all four sides with a low value in the center.

The deepest part of the trough in the center is relatively narrow with respect to the folding wave numbers at the edges. However, it is somewhat broader than the peak in the spectrum at the lower left.

The result of convolution

The image in the upper right shows the result of convolving the space domain surface in the upper left with the convolution operator in the upper center. This output space domain surface has a green square area in the center that is at the same level as the green background. In this case, green represents an elevation of 0, which is about midway between the lowest elevation (black) and the highest elevation (white).

Positive and negative fences

Surrounding the green square is a yellow and white fence representing very high elevations. Surrounding that fence is a black and blue fence, representing very low elevations consisting of large negative values.

Thus, as you move from the outside to the inside of the square in the output surface, the elevation goes from a background level of zero, to a large negative value, followed immediately by a large positive value, followed by zero.

Edge detection

This is one form of edge detection. The edges of the square in the input surface have been emphasized and the flat portion of the input surface has been deemphasized in the convolution output.

Wavenumber spectrum of the convolution output

The wavenumber spectrum of the output from the convolution operation is shown in the lower right. The spectrum indicates that this surface is made up mostly of wavenumber components having mid range to high values.

If you are familiar with digital signal processing, you will know that in order for a space (*or time*) function to contain very rapid changes in value (*such as the elevation changes at the fences described above*) the function must contain significant high wavenumber (*or frequency*) components. That appears to be the case here indicated by the red areas on the four sides of the wavenumber spectrum.

Although this spectrum was produced by convolution in the space domain followed by a 2D Fourier transform on the convolution output, you should be able to see that the shape of the spectrum on the bottom right approximates the product of the spectrum of the original surface on the bottom left and the spectral response of the convolution operator in the bottom center.

Thus, the same results could have been produced using multiplication in the wavenumber domain followed by an inverse Fourier transform to produce the space domain result. Convolution in the space domain is equivalent to multiplication in the wavenumber domain and vice versa.

Another interesting application that I can demonstrate online is using 2D Fourier transforms to hide secret trademarks and watermarks in images. The purpose of a hidden trademark or watermark is for the owner of the image to be able to demonstrate that the image may have been used inappropriately by someone else. Once again, this application can be satisfied by treating the space domain data as purely real. I plan to demonstrate how this is done in a future module.

Summary

I began by explaining how the space domain and the wavenumber domain in two-dimensional analysis are analogous to the time domain and the frequency domain in one-dimensional analysis.

Then I introduced you to some practical examples showing how 2D Fourier transforms and wavenumber spectra can be useful in solving engineering problems involving antenna arrays.

What's next?

In [Part 2](#) of this two-part series, I will provide and explain a Java class that can be used to perform forward and inverse 2D Fourier transforms, and can also be used to shift the wavenumber origin from the upper left to the center for a more pleasing plot of the wavenumber spectrum.

In addition, I will provide and explain a program that is used to:

- Test the forward and inverse 2D Fourier transforms to confirm that the code is correct and that the transformations behave as they should
- Produce wavenumber spectra for simple surfaces to help the student gain a feel for the relationship that exists between the space domain and the wavenumber domain

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1490-2D Fourier Transforms using Java
- File: Java1490.htm
- Published: 07/12/05

Learn how the space domain and the wavenumber domain in two-dimensional analysis are analogous to the time domain and the frequency domain in one-dimensional analysis. Learn about some practical examples showing how 2D Fourier transforms and wavenumber spectra can be useful in solving engineering problems involving antenna arrays.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1491-2D Fourier Transforms using Java, Part 2

Examine the code for a Java class that can be used to perform forward and inverse 2D Fourier transforms on 3D surfaces in the space domain. Learn how the 2D Fourier transform behaves for a variety of different sample surfaces in the space domain.

Revised: Wed Oct 21 16:16:44 CDT 2015

This page is included in the following book: [Digital Signal Processing - DSP](#)

Table of contents

- [Table of contents](#)
- [Preface](#)
 - [Two separate programs](#)
 - [Using the class named ImgMod30](#)
 - [Digital signal processing.\(DSP\)](#)
 - [Will use in subsequent modules](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General discussion](#)
 - [The space domain](#)
 - [A purely real space domain](#)
- [Preview](#)
- [Sample programs](#)
 - [The class named ImgMod30](#)

- [The xform2D method](#)
- [The inverseXform2D method](#)
- [The shiftOrigin method](#)
- [The program named ImgMod31](#)
 - [Command line parameters](#)
 - [Forward and inverse Fourier transforms](#)
 - [Fourteen cases](#)
 - [A stack of output images](#)
 - [Numeric output](#)
 - [How to use the program named ImgMod31](#)
 - [Program code for ImgMod31](#)
- [Code and graphic output for fourteen example cases](#)
 - [Case 0](#)
 - [Case 1](#)
 - [Case 2](#)
 - [Case 3](#)
 - [Case 4](#)
 - [Case 5](#)
 - [Case 6](#)
 - [Case 7](#)
 - [Case 8](#)
 - [Case 9](#)
 - [Case 10](#)
 - [Case 11](#)
 - [Case 12](#)
 - [Case 13](#)
- [Run the program](#)
- [Summary](#)
- [Complete program listings](#)
- [Miscellaneous](#)

Preface

This is the second module in a two-part series. The first part published earlier was titled [Java1490-2D Fourier Transforms using Java, Part 1](#). In this module, I will teach you how to perform two-dimensional (2D) Fourier transforms using Java. I will

- Explain the conceptual and computational aspects of 2D Fourier transforms
- Explain the relationship between the space domain and the [wavenumber](#) domain
- Provide sufficient background information that you will be able to appreciate the importance of the 2D Fourier transform
- Provide Java software to perform 2D Fourier transforms
- Provide Java software to test and exercise that capability

Two separate programs

I will present and explain two separate programs. One program consists of a single class named **ImgMod30**. The purpose of this class is to satisfy the computational requirements for forward and inverse 2D Fourier transforms. This class also provides a method for rearranging the spectral data into a more useful format for plotting. The second program named **ImgMod31** will be used to test the 2D Fourier transform class, and also to illustrate the use of 2D Fourier transforms for some well known sample surfaces.

A third class named **ImgMod29** will be used to display various 3D surfaces resulting from the application of the 2D Fourier transform. I explained this class in an earlier module titled [Plotting 3D Surfaces using Java](#).

Using the class named ImgMod30

The 2D Fourier transform class couldn't be easier to use. To perform a forward transform execute a statement similar to the following:

```
ImgMod30.xform2D(spatialData, realSpect,  
                 imagSpect, amplitudeSpect);
```

The first parameter in the above statement is a reference to an array object containing the data to be transformed. The other three parameters refer to array objects that will be populated with the results of the transform.

To perform an inverse transform execute a statement similar to the following:

```
ImgMod30.inverseXform2D(realSpect, imagSpect,  
                        recoveredSpatialData);
```

The first two parameters in the above statement refer to array objects containing the complex spectral data to be transformed. The third parameter refers to an array that will be populated with the results of the inverse transform.

To rearrange the spectral data for plotting, execute a statement similar to the following where the parameter refers to an array object containing the spectral data to be rearranged.

```
double[][] shiftedRealSpect =  
    ImgMod30.shiftOrigin(realSpect);
```

Digital signal processing (DSP)

This module will cover some technically difficult material in the general area of Digital Signal Processing, or DSP for short. As usual, the better prepared you are, the more likely you are to understand the material. For example, it would be well for you to already understand the one-dimensional Fourier transform before tackling the 2D Fourier transform. If you don't already have that knowledge, you can learn about one-dimensional Fourier transforms by studying the following **modules** :

- [Java1478 Fun with Java, How and Why Spectral Analysis Works](#)
- [Java1482 Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm](#)
- [Java1483 Spectrum Analysis using Java, Frequency Resolution versus Data Length](#)
- [Java1484 Spectrum Analysis using Java, Complex Spectrum and Phase Angle](#)

- [Java1485 Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain](#)
- [Java1486 Fun with Java, Understanding the Fast Fourier Transform \(FFT\) Algorithm](#)

In addition, I strongly recommend that you study [Java1490-2D Fourier Transforms using Java, Part 1](#) before embarking on this part.

Will use in subsequent modules

The 2D Fourier transform has many uses. I will use the 2D Fourier transform in several future modules involving such diverse topics as:

- Processing image pixels in the wavenumber domain
- Advanced steganography (*hiding messages in images*)
- Hiding watermarks and trademarks in images

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1.](#) Two views of the same wavenumber spectrum.
- [Figure 2.](#) Numeric output.
- [Figure 3.](#) How to use the program named ImgMod31.
- [Figure 4.](#) An example test surface plot.
- [Figure 5.](#) An impulse in space.
- [Figure 6.](#) A displaced impulse in space.
- [Figure 7.](#) A box on the diagonal in space.
- [Figure 8.](#) Graphic output for Case 3.
- [Figure 9.](#) Graphic output for Case 4.

- [Figure 10.](#) Graphic output for Case 5.
- [Figure 11.](#) Graphic output for Case 6.
- [Figure 12.](#) Graphic output for Case 7.
- [Figure 13.](#) Graphic output for Case 9.
- [Figure 14.](#) Graphic output for Case 11.
- [Figure 15.](#) Graphic output for Case 12.
- [Figure 16.](#) Graphic output for Case 13.

Listings

- [Listing 1.](#) Beginning of the class named ImgMod30.
- [Listing 2.](#) The remainder of the xform2D method.
- [Listing 3.](#) The inverseXform2D method.
- [Listing 4.](#) The shiftOrigin method code.
- [Listing 5.](#) Beginning of the class named ImgMod31.
- [Listing 6.](#) Create and save the test surface.
- [Listing 7.](#) Display the test surface.
- [Listing 8.](#) Perform the forward Fourier transform.
- [Listing 9.](#) Display unshifted amplitude spectrum.
- [Listing 10.](#) Shift the origin and display the results.
- [Listing 11.](#) Perform an inverse transform.
- [Listing 12.](#) Display the result of the inverse transform.
- [Listing 13.](#) Display some numeric results.
- [Listing 14.](#) Beginning of the getSpatialData method.
- [Listing 15.](#) Code for Case 0.
- [Listing 16.](#) Code for Case 1.
- [Listing 17.](#) Code for Case 2.
- [Listing 18.](#) Code for Case 7.
- [Listing 19.](#) Code for Case 8.
- [Listing 20.](#) The end of the getSpatialData method.
- [Listing 21.](#) ImgMod30.java.
- [Listing 22.](#) ImgMod31.java.

General discussion

The space domain

In [Part 1](#) of this series, I extended the concept of the Fourier transform from the time domain into the space domain. I pointed out that while the time domain is one-dimensional, the space domain is three-dimensional. However, in order to keep the complexity of this module in check, we will assume that space is only two-dimensional. This will serve us well later for such tasks as image processing.

(Three-dimensional Fourier transforms are beyond the scope of this module. I will write a module on using Fourier transforms in three-dimensional space later if I have the time.)

A purely real space domain

Although the space domain can be (*and often is*) complex, many interesting problems, (*such as photographic image processing*) can be dealt with under the assumption that the space domain is purely real. We will make that assumption in this module. This assumption will allow us to simplify our computations when performing the 2D Fourier transform to transform our data from the space domain into the wavenumber domain.

Preview

I will present and explain two complete Java programs in this module. The first program is a single class named **ImgMod30**, which provides the capability to perform forward and inverse Fourier transforms on three-dimensional surfaces. In addition, the class provides a method that can be used to reformat the wavenumber spectrum to make it more suitable for display.

The second program is named **ImgMod31**. This is an executable program whose purpose is to exercise and to test the **ImgMod30** class using several examples for which the results should already be known.

Sample programs

The class named `ImgMod30`

This class provides 2D Fourier transform capability that can be used for image processing and other purposes. The class provides three static methods:

- **`xform2D`** : Performs a forward 2D Fourier transform on a purely real surface described by a 2D array of **`double`** values in the space domain to produce a complex spectrum in the wavenumber domain. The method returns the real part, the imaginary part, and the amplitude spectrum, each in its own 2D array of **`double`** values.
- **`inverseXform2D`** : Performs an inverse 2D Fourier transform from the complex wavenumber domain into the real space domain using the real and imaginary parts of the wavenumber spectrum as input. Returns the surface in the space domain as a 2D array of **`double`** values. Assumes that the real and imaginary parts in the wavenumber domain are consistent with a purely real surface in the space domain, and does not return an imaginary surface for the space domain
- **`shiftOrigin`** : The wavenumber spectrum produced by **`xform2D`** has its origin in the top-left corner with the Nyquist folding wave numbers near the center. This is not a very suitable format for visual analysis. This method rearranges the data to place the origin at the center with the Nyquist folding wave numbers along the edges.

The class was tested using JDK 1.8 and Windows 7. The class uses the static import capability that was introduced in J2SE 5.0. Therefore, it should not compile using earlier versions of the compiler.

The `xform2D` method

The beginning of the class and the beginning of the static method named **`xform2D`** is shown in [Listing 1](#).

This method computes a forward 2D Fourier transform from the space domain into the wavenumber domain. The number of points produced for the wavenumber domain matches the number of points received for the space

domain in both dimensions. Note that the input data must be purely real. In other words, the program assumes that there are no imaginary values in the space domain. Therefore, this is not a general purpose 2D complex-to-complex transform.

Listing 1. Beginning of the class named ImgMod30.

```
class ImgMod30{
    static void xform2D(double[][] inputData,
                        double[][] realOut,
                        double[][] imagOut,
                        double[][] amplitudeOut){

        int height = inputData.length;
        int width = inputData[0].length;

        System.out.println("height = " + height);
        System.out.println("width = " + width);
```

Parameters

The first parameter is a reference to a 2D **double** array object containing the data to be transformed. The remaining three parameters are references to 2D **double** array objects of the same size that will be populated with the following transform results:

- The real part
- The imaginary part
- The amplitude (*square root of sum of squares of the real and imaginary parts*)

[Listing 1](#) also determines and displays the dimensions of the incoming 2D array of data to be transformed.

I won't bore you with the details as to how and why the 2D Fourier transform does what it does. Neither will I bore you with the details of the code that implements the 2D Fourier transform. If you understand the material that I have [previously published](#) on Fourier transforms in one dimension, this code and these concepts should be a straightforward extension from one dimension to two dimensions.

The remainder of the `xform2D` method

The remainder of the `xform2D` method is shown in [Listing 2](#). Note that it was necessary to sacrifice indentation in order to force these very long equations to be compatible with this narrow publication format and still be somewhat readable.

Listing 2. The remainder of the `xform2D` method.

```
//Two outer loops iterate on output data.
for(int yWave = 0;yWave < height;yWave++){
    for(int xWave = 0;xWave < width;xWave++){
        //Two inner loops iterate on input data.
        for(int ySpace = 0;ySpace < height;
            ySpace++)
        {
            for(int xSpace = 0;xSpace < width;
                xSpace++)
            {
                //Compute real, imag, and amplitude.
                realOut[yWave][xWave] +=
                (inputData[ySpace][xSpace]*cos(2*PI*((1.0*
```

Listing 2. The remainder of the xform2D method.

```
xWave*xSpace/width)+
(1.0*yWave*ySpace/height))))
/sqrt(width*height);

imagOut[yWave][xWave] -=
(inputData[ySpace][xSpace]*sin(2*PI*
((1.0*xWave*
xSpace/width) + (1.0*yWave*ySpace/height))))
/sqrt(width*height);

amplitudeOut[yWave][xWave] =
sqrt(
    realOut[yWave][xWave] * realOut[yWave][xWave]
+
    imagOut[yWave][xWave] * imagOut[yWave]
[xWave]);
    }//end xSpace loop
    }//end ySpace loop
    }//end xWave loop
    }//end yWave loop
    }//end xform2D method
```

The inverseXform2D method

The **inverseXform2d** method is shown in its entirety in [Listing 3](#). This method computes an inverse 2D Fourier transform from the complex wavenumber domain into the real space domain. The number of points produced for the space domain matches the number of points received for the wavenumber domain in both dimensions.

This method assumes that the inverse transform will produce purely real values in the space domain. Therefore, in the interest of computational efficiency, the method does not compute the imaginary output values. Therefore, this is not a general purpose 2D complex-to-complex transform.

For correct results, the input complex data must match that obtained by performing a forward transform on purely real data in the space domain.

Once again it was necessary to sacrifice indentation to force this very long equation to be compatible with this narrow publication format and still be readable.

Listing 3. The inverseXform2D method.

```
static void inverseXform2D(double[][] real,
                           double[][] imag,
                           double[][] dataOut)
{
    int height = real.length;
    int width = real[0].length;

    System.out.println("height = " + height);
    System.out.println("width = " + width);

    //Two outer loops iterate on output data.
    for(int ySpace = 0;ySpace < height;ySpace++)
    {
        for(int xSpace = 0;xSpace < width;
              xSpace++)
        {
            //Two inner loops iterate on input data.
            for(int yWave = 0;yWave < height;
                  yWave++)
            {
                for(int xWave = 0;xWave < width;
                      xWave++)
```

Listing 3. The inverseXform2D method.

```
{
//Compute real output data.
dataOut[ySpace][xSpace] +=
    (real[yWave][xWave]*cos(2*PI*((1.0 * xSpace*
xWave/width) + (1.0*ySpace*yWave/height))) -
    imag[yWave][xWave]*sin(2*PI*((1.0 * xSpace*
xWave/width) + (1.0*ySpace*yWave/height))))
    /sqrt(width*height);
    }//end xWave loop
    }//end yWave loop
    }//end xSpace loop
    }//end ySpace loop
} //end inverseXform2D method
```

Parameters

This method requires three parameters. The first two parameters are references to 2D arrays containing the real and imaginary parts of the complex wavenumber spectrum that is to be transformed into a surface in the space domain.

The third parameter is a reference to a 2D array of the same size that will be populated with the transform results.

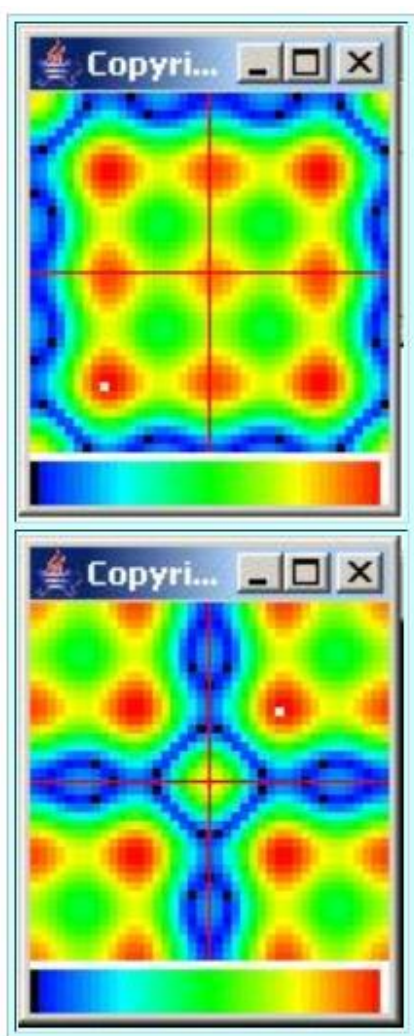
The shiftOrigin method

This method deserves some explanation. The reason that this method is needed is illustrated by [Figure 1](#).

Two views of the same wavenumber spectrum

Both of the images in [Figure 1](#) represent the same wavenumber spectrum, but they are plotted against different coordinate systems.

Figure 1. Two views of the same wavenumber spectrum.



How the wavenumber spectrum is actually computed

The top image shows how the wavenumber spectrum is actually computed.

The wavenumber spectrum is computed covering an area of wavenumber space with the 0,0 origin in the top-left corner. The computation extends to twice the Nyquist folding wave number along each axis.

Computationally sound but not visually pleasing

While this format is computationally sound, it isn't what most of us are accustomed to seeing in plots in wavenumber space. Rather, we are accustomed to seeing wavenumber spectra plotted with the 0,0 origin at the center.

The wavenumber spectrum is periodic

Knowing that the area of wavenumber space shown in the top image of [Figure 1](#) covers one complete period of a periodic surface, the **shiftOrigin** method rearranges the results (*for display purposes only*) to that shown in the bottom image in [Figure 1](#). The origin is at the center in the bottom image of [Figure 1](#). The edges of the lower image in [Figure 1](#) are the Nyquist folding wave numbers.

The shiftOrigin method code

The **shiftOrigin** method is shown in its entirety in [Listing 4](#). Although this method is rather long, it is also completely straightforward. Therefore, it shouldn't require a further explanation. You may be able to develop a much shorter algorithm for accomplishing the same task.

Listing 4. The shiftOrigin method code.

```
//Method to shift the wavenumber origin and
// place it at the center for a more visually
// pleasing display. Must be applied
// separately to the real part, the imaginary
// part, and the amplitude spectrum for a
// wavenumber spectrum.
static double[][] shiftOrigin(double[][] data)
{
    int numberOfRows = data.length;
    int numberOfCols = data[0].length;
    int newRows;
    int newCols;

    double[][] output =
        new double[numberOfRows]
[numberOfCols];

    //Must treat the data differently when the
    // dimension is odd than when it is even.

    if(numberOfRows%2 != 0){//odd
        newRows = numberOfRows +
            (numberOfRows +
1)/2;
    }else{//even
        newRows = numberOfRows + numberOfRows/2;
    }//end else

    if(numberOfCols%2 != 0){//odd
        newCols = numberOfCols +
            (numberOfCols +
1)/2;
    }else{//even
        newCols = numberOfCols + numberOfCols/2;
    }//end else
```

Listing 4. The shiftOrigin method code.

```
//Create a temporary working array.
double[][] temp =
    new double[newRows]
[newCols];

//Copy input data into the working array.
for(int row = 0;row < numberOfRows;row++){
    for(int col = 0;col < numberOfCols;col++){
        temp[row][col] = data[row][col];
    }//col loop
}//row loop

//Do the horizontal shift first
if(numberOfCols%2 != 0){//shift for odd

    //Slide leftmost (numberOfCols+1)/2
columns
    // to the right by numberOfCols columns
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
            col < (numberOfCols+1)/2;col++)
        {
            temp[row][col + numberOfCols] =
temp[row]
[col];
        }//col loop
    }//row loop

    //Now slide everything back to the left by
    // (numberOfCols+1)/2 columns
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
            col < numberOfCols;col++)
        {
            temp[row][col] =
```

Listing 4. The shiftOrigin method code.

```
        temp[row][col+(numberOfCols +
1)/2];
    }//col loop
}//row loop

}else{//shift for even
    //Slide leftmost (numberOfCols/2) columns
    // to the right by numberOfCols columns.
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
            col < numberOfCols/2;col++)
        {
            temp[row][col + numberOfCols] =
                temp[row]
[col];
        }//col loop
    }//row loop

    //Now slide everything back to the left by
    // numberOfCols/2 columns
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
            col < numberOfCols;col++)
        {
            temp[row][col] =
                temp[row][col +
numberOfCols/2];
        }//col loop
    }//row loop
}//end else

//Now do the vertical shift
if(numberOfRows%2 != 0){//shift for odd
    //Slide topmost (numberOfRows+1)/2 rows
    // down by numberOfRows rows.
    for(int col = 0;col < numberOfCols;col++){
```

Listing 4. The shiftOrigin method code.

```
        for(int row = 0;
            row < (numberOfRows+1)/2;row++)
    {
        temp[row + numberOfRows][col] =
            temp[row]
[col];
    }//row loop
} //col loop

//Now slide everything back up by
// (numberOfRows+1)/2 rows.
for(int col = 0;col < numberOfCols;col++){
    for(int row = 0;
        row < numberOfRows;row++)
    {
        temp[row][col] =
            temp[row+(numberOfRows + 1)/2]
[col];
    }//row loop
} //col loop

}else{//shift for even
    //Slide topmost (numberOfRows/2) rows down
    // by numberOfRows rows
    for(int col = 0;col < numberOfCols;col++){
        for(int row = 0;
            row < numberOfRows/2;row++)
        {
            temp[row + numberOfRows][col] =
                temp[row]
[col];
        }//row loop
    } //col loop

    //Now slide everything back up by
    // numberOfRows/2 rows.
```

Listing 4. The shiftOrigin method code.

```
        for(int col = 0;col < numberOfCols;col++){
            for(int row = 0;
                row < numberOfRows;row++)
        {
            temp[row][col] =
                temp[row + numberOfRows/2]
[col];
        }//row loop
    }//col loop
} //end else

//Shifting of the origin is complete. Copy
// the rearranged data from temp to output
// array.
for(int row = 0;row < numberOfRows;row++){
    for(int col = 0;col < numberOfCols;col++){
        output[row][col] = temp[row][col];
    }//col loop
} //row loop

    return output;
} //end shiftOrigin method

} //end class ImgMod30
```

End of the ImgMod30 class

[Listing 4](#) also signals the end of the class definition for the class named ImgMod30.

The program named ImgMod31

The purpose of this program is to exercise and to test the 2D Fourier transform methods and the axis shifting method provided by the class named `ImgMod30`.

Command line parameters

The **main** method in this class reads two command line parameters and uses them to select:

- A specific case involving a particular 3D input surface in the space domain.
- A specific display format.

Forward and inverse Fourier transforms

The program performs a 2D Fourier transform on that surface followed by an inverse 2D Fourier transform. Six different plots are produced in this process showing different aspects of the transform and the inverse transform.

Fourteen cases

There are 14 different cases built into the program with case numbers ranging from 0 to 13 inclusive. Each of the cases is designed such that the results of the analysis should be known in advance by a person familiar with 2D Fourier analysis and the wavenumber domain. Thus, these cases can be used to confirm that the transform code was properly written.

The cases are also designed to illustrate the impact of various space domain characteristics on the wavenumber spectrum. This information will be useful later when analyzing the results of performing 2D transforms on photographic images for example.

A stack of output images

Each time the program is run, it produces a stack of six output images in the top-left corner of the screen. A brief description of each of the output images is provided in the following list. The top-to-bottom order of the stack is:

1. Space domain output of inverse Fourier transform. Compare with original input in 6 below.
2. Amplitude spectrum in wavenumber domain with shifted origin. Compare with 5 below.
3. Imaginary wavenumber spectrum with shifted origin.
4. Real wavenumber spectrum with shifted origin.
5. Amplitude spectrum in wavenumber domain without shifted origin. Compare with 2 above.
6. Space domain input data. Compare with 1 above.

To view the images near the bottom of the stack, you must physically move those on top to get them out of the way.

Numeric output

In addition, the program produces some numeric output on the command line screen that may be useful in confirming the validity of the forward and inverse transform processes. [Figure 2](#) shows an example of the numeric output.

Figure 2. Numeric output.

Figure 2. Numeric output.

```
height = 41
width = 41
height = 41
width = 41

2.0          1.9999999999999916
0.50000000000000002  0.49999999999999845
0.49999999999999956  0.4999999999999923
1.7071067811865475   1.7071067811865526
0.2071067811865478   0.20710678118654233
0.20710678118654713  0.20710678118655435
1.0            1.00000000000000064
-0.4999999999999997 -0.49999999999999484
-0.50000000000000003 -0.4999999999999965
```

(Note that I manually inserted some and spaces line breaks in [Figure 2](#) to cause the numeric values to line up in columns so as to be more readable.)

The size of the surfaces

The first two lines of numeric output in [Figure 2](#) show the size of the spatial surface for the forward transform. The second two lines show the size of the wavenumber surface for the inverse transform.

The quality of the transform process

The remaining nine lines indicate something about the quality of the forward and inverse transforms in terms of the ability of the inverse transform to

replicate the original spatial surface. These lines also indicate something about the correctness of the overall scaling from original input to final output.

Matching pairs of values

Each of the last nine lines contains a pair of values. The first value is a sample from the original spatial surface. The second value is a sample from the corresponding location on the spatial surface produced by performing an inverse transform on the wavenumber spectrum. The two values in each pair of values should match. If they match, this indicates the probability of a valid result.

(Note however that this is a very small sampling of the values that make up the original and replicated spatial data and problems could arise in areas that are not included in this small sample.)

The match is very good in the example shown above. This example is from Case #12.

How to use the program named ImgMod31

Usage information for the program is shown in [Figure 3](#).

Figure 3. How to use the program named ImgMod31.

Figure 3. How to use the program named ImgMod31.

Usage:

```
java ImgMod31 CaseNumber DisplayType
```

CaseNumber ranges from 0 to 13 inclusive.

DisplayType ranges from 0 to 2 inclusive.

If a case number is not provided, Case 2 will be run by default.

If a display type is not provided, display type 1 will be used by default.

A description of each case is provided by the comments in this program. In addition, each case will be discussed in detail in this module.

See **ImgMod29** in the earlier module titled [Plotting 3D Surfaces using Java](#) for a definition of **DisplayType**.

You can terminate the program by clicking on the close button on any of the display frames produced by the program.

Program code for ImgMod31

The beginning of the class and the beginning of the **main** method is shown in [Listing 5](#).

Listing 5. Beginning of the class named ImgMod31.

```
class ImgMod31{

    public static void main(String[] args){
        int switchCase = 2;//default
        int displayType = 1;//default
        if(args.length == 1){
            switchCase = Integer.parseInt(args[0]);
        }else if(args.length == 2){
            switchCase = Integer.parseInt(args[0]);
            displayType = Integer.parseInt(args[1]);
        }else{
            System.out.println("Usage: java ImgMod31 "
                               + "CaseNumber
DisplayType");
            System.out.println(
                "CaseNumber from 0 to 13
inclusive.");
            System.out.println(
                "DisplayType from 0 to 2
inclusive.");
            System.out.println("Running case "
                               + switchCase + " by
default.");
            System.out.println("Running DisplayType "
                               + displayType + " by
default.");
        }//end else
    }
```

The code in [Listing 5](#) gets the input parameters and uses them to set the case and the display format. A default case and a default display format are used if this information is not provided by the user.

Create and save the test surface

[Listing 6](#) calls the method named **getSpatialData** to create a test surface that matches the specified case. This surface will be used for testing the transform process.

Listing 6. Create and save the test surface.

```
int rows = 41;
int cols = 41;

double[][] spatialData =
getSpatialData(switchCase, rows, cols);
```

I will discuss the method named **getSpatialData** in detail later. For now, just assume that the 2D array object referred to by **spatialData** contains the test surface when this method returns.

Display the test surface

[Listing 7](#) instantiates an object of the class named **ImgMod29** to display the test surface in the display format indicated by the value of **displayType**.

Listing 7. Display the test surface.

Listing 7. Display the test surface.

```
new  
ImgMod29(spatialData, 3, false, displayType);
```

The value of false in the third parameter indicates that the axes should not be displayed.

(See the module titled [Plotting 3D Surfaces using Java](#) for an explanation of the second parameter. Basically, this parameter is used to control the overall size of the plot on the screen.)

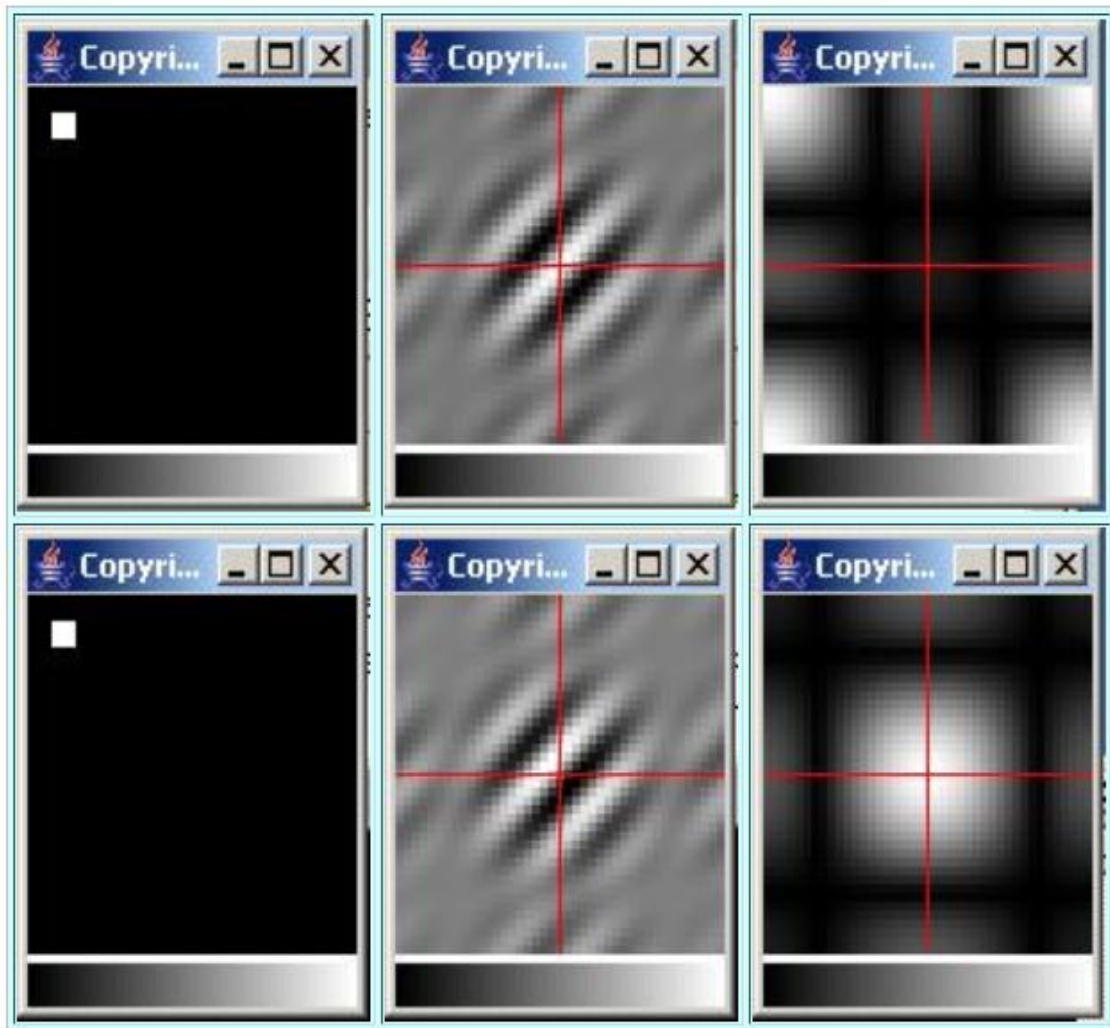
An example test surface plot

The top-left image in [Figure 4](#) is an example of the output produced by the code in [Listing 7](#) for a **displayType** value of 0.

([Figure 4](#) shows the grayscale format. See the module titled [Plotting 3D Surfaces using Java](#) for an explanation of the three available non-logarithmic display formats.)

Figure 4. An example test surface plot.

Figure 4. An example test surface plot.



[Figure 4](#) shows the results for a test surface **switchCase** value of 2. I will discuss the particulars of this case in detail later.

Perform the forward Fourier transform

[Listing 8](#) performs the forward Fourier transform to transform the test surface into the wavenumber domain.

Listing 8. Perform the forward Fourier transform.

```
double[][] realSpect = //Real part
                      new double[rows]
[cols];
double[][] imagSpect = //Imaginary part
                      new double[rows]
[cols];
double[][] amplitudeSpect = //Amplitude
                             new double[rows]
[cols];

ImgMod30.xform2D(spatialData, realSpect,
imagSpect, amplitudeSpect);
```

Prepare array objects to receive the transform results

[Listing 8](#) begins by preparing some array objects to receive the transform results. The forward transform receives an incoming surface array and returns the real and imaginary parts of the complex wavenumber spectrum along with the amplitude spectrum by populating three array objects passed as parameters to the method.

Perform the forward transform

Then [Listing 8](#) calls the static **xform2D** method of the **ImgMod30** class to perform the forward transform, returning the results by way of the parameters to the method.

Display unshifted amplitude spectrum

The top-right image in [Figure 4](#) is an example of the type of display produced by the code in [Listing 9](#). This is a plot of the amplitude spectrum without the wavenumber origin being shifted to place it at the center.

(The wavenumber origin is in the top-left corner of the top-right image in [Figure 4](#).)

This image also shows the result of passing true as the third parameter causing the red axes to be plotted on top of the spectral data.

Listing 9. Display unshifted amplitude spectrum.

```
new ImgMod29(amplitudeSpect,3,true,  
displayType);
```

Need to shift the origin for display

The top-right image in [Figure 4](#) is in a format that is not particularly good for viewing. In particular, the origin is at the top-left corner. The horizontal Nyquist folding wavenumber is near the horizontal center of the plot. The vertical Nyquist folding wave number is near the vertical center of the plot. It is much easier for most people to understand the plot when the wavenumber origin is shifted to the center of the plot with the Nyquist folding wave numbers at the edges of the plot.

The method named **shiftOrigin** can be used to rearrange the data and shift the origin to the center of the plot.

Shift the origin and display the results

[Listing 10](#) shifts the origin to the center of the plot and displays:

- The real part of the shifted spectrum
- The imaginary part of the shifted spectrum
- The amplitude of the shifted spectrum

The axes are displayed in all three cases.

Listing 10. Shift the origin and display the results.

Listing 10. Shift the origin and display the results.

```
double[][] shiftedRealSpect =  
ImgMod30.shiftOrigin(realSpect);  
    new ImgMod29(shiftedRealSpect, 3, true,  
displayType);  
  
double[][] shiftedImagSpect =  
ImgMod30.shiftOrigin(imagSpect);  
    new ImgMod29(shiftedImagSpect, 3, true,  
displayType);  
  
double[][] shiftedAmplitudeSpect =  
ImgMod30.shiftOrigin(amplitudeSpect);  
    new ImgMod29(shiftedAmplitudeSpect, 3, true,  
displayType);
```

Example displays

Examples of the displays produced by the code in [Listing 10](#) are shown in [Figure 4](#). The surface being transformed is shown in the top-left in [Figure 4](#) and the result of the inverse transform (*discussed later*) is shown in the bottom-left in [Figure 4](#).

The real part of the shifted wavenumber spectrum is shown in the image in the top-center of [Figure 4](#). The imaginary part of the shifted wavenumber spectrum is shown in the bottom-center of [Figure 4](#).

The unshifted amplitude spectrum is shown in the top-right in [Figure 4](#) and the shifted amplitude spectrum is shown in the bottom-right image in [Figure 4](#). The origin has been shifted to the center in the three cases shown in the top-center, bottom-center, and bottom-right.

(It would probably be constructive for you to compare the two rightmost images in [Figure 4](#) in order to appreciate the result of shifting the origin to the center.)

Perform an inverse transform

[Listing 11](#) performs an inverse Fourier transform to transform the complex wavenumber surface into a real surface in the space domain. Ideally, the result should exactly match the space domain surface that was transformed into the wavenumber domain in [Listing 8](#). However, because of small arithmetic errors that accumulate in the forward and inverse transform computations, it is unusual for an exact match to be achieved.

Listing 11. Perform an inverse transform.

```
double[][] recoveredSpatialData =  
    new double[rows]  
[cols];  
  
ImgMod30.inverseXform2D(realSpect, imagSpect,  
recoveredSpatialData);
```

Prepare an array object to store the results

[Listing 11](#) begins by preparing an array object to store the results of the inverse transform process.

Call the `inverseXform2D` method

Then [Listing 11](#) calls the **`inverseXform2D`** method to transform the complex wavenumber spectrum into a real space function. The **`inverseXform2D`** method requires the real and imaginary parts of the complex wavenumber spectrum as input parameters.

(Note that these are the original real and imaginary parts of the complex wavenumber spectrum. They are not the versions for which the origin has been shifted for display purposes.)

The **`inverseXform2D`** method also receives an incoming array object in which to store the real result of the transform process.

Display the result of the inverse transform

Finally, [Listing 12](#) displays the result of the inverse transform as a surface in the space domain. This surface should compare favorably with the original surface that was transformed into the wavenumber domain in [Listing 8](#)

Listing 12. Display the result of the inverse transform.

Listing 12. Display the result of the inverse transform.

```
new ImgMod29(recoveredSpatialData,3,false,  
displayType);
```

The output produced by [Listing 12](#) is shown in the lower-left image in [Figure 4](#). Compare this with the input surface shown in the top-left image in [Figure 4](#). As you can see, they do compare favorably. In fact, they appear to be identical in this grayscale plotting format. We will see later that when a more sensitive plotting format is used, small differences in the two may become apparent.

Display some numeric results

As discussed earlier, the code in [Listing 13](#) samples and displays a few corresponding points on the original surface and the surface produced by the inverse transform process. The results can be used to evaluate the overall quality of the process as well as the correctness of the overall scaling.

Listing 13. Display some numeric results.

Listing 13. Display some numeric results.

```
        for(int row = 0;row < 3;row++){
            for(int col = 0;col < 3;col++){
                System.out.println(
                    spatialData[row][col] + " " +
                    recoveredSpatialData[row][col] + "
");
            }//col
        }//row
    }//end main
```

Each line of output text contains two values, and ideally the two values should be exactly the same. Realistically, because of small computational errors in the transform and inverse transform process, it is unlikely that the two values will be exactly the same except in computationally trivial cases. Unless the two values are very close, however, something probably went wrong in the transform process and the results should not be trusted.

[Listing 13](#) also signals the end of the **main** method.

What do we know so far?

Now we know how to use the **ImgMod30** class and the **ImgMod29** class to:

- Transform a purely real 3D surface from the space domain into the wavenumber domain
- Transform a complex wavenumber spectrum into a purely real surface in the space domain
- Shift the origin of the real, imaginary, and amplitude wavenumber spectral parts to convert the data into a format that is more suitable for plotting
- Plot 3D surfaces in both domains

It is time for us to take a look at the method named **getSpatialData** that can be used to create any of fourteen standard surfaces in the space domain.

The getSpatialData method

This method constructs and returns a specific 3D surface in a 2D array of type **double** . The surface is identified by the value of an incoming parameter named **switchCase** . There are 14 possible cases. The allowable values for **switchCase** range from 0 through 13 inclusive.

The other two input parameters specify the size of the surface that will be produced in units of rows and columns.

The code for the getSpatialData method

The **getSpatialData** method begins in [Listing 14](#).

Listing 14. Beginning of the getSpatialData method.

```
private static double[][] getSpatialData(
    int switchCase,int rows,int cols)
{
    double[][] spatialData =
        new double[rows]
[cols];
    switch(switchCase){
```


[Listing 14](#) begins by creating a 2D array object of type **double** in which to store the surface.

Then [Listing 14](#) shows the beginning of a **switch** statement that will be used to select the code to create a surface that matches the value of the incoming parameter named **switchCase** .

Code and graphic output for fourteen example cases

Case 0

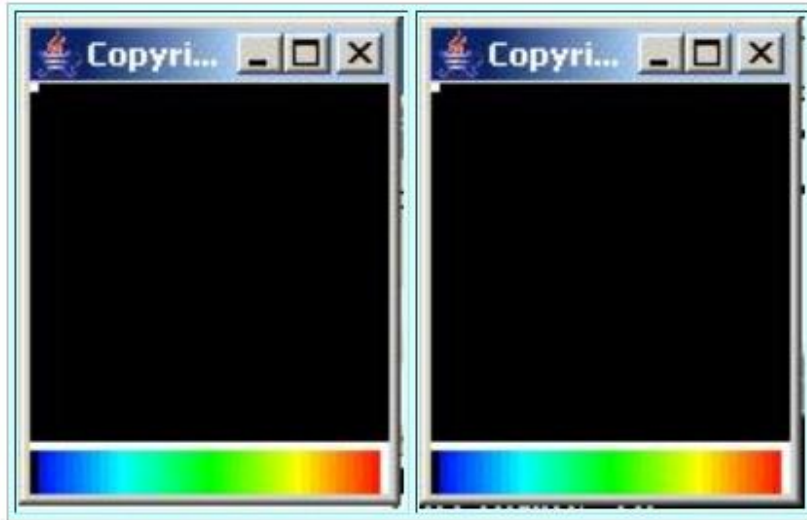
[Listing 15](#) shows the code that is executed for a value of **switchCase** equal to 0.

Listing 15. Code for Case 0.

```
case 0:
    spatialData[0][0] = 1;
    break;
```

This case places a single non-zero point at the origin in the space domain. The origin is at the top-left corner. The surface produced by this case is shown in the leftmost image in [Figure 5](#). The non-zero value can be seen as the small white square in the top-left corner. In signal processing terminology, this point can be viewed as an impulse in space. It is well known that such an impulse produces a flat spectrum in wavenumber space.

Figure 5. An impulse in space.



The output surface

The rightmost image in [Figure 5](#) shows the result of:

- Performing a forward Fourier transform on the surface in the leftmost image
- Performing an inverse Fourier transform on the complex wavenumber spectrum produced by the forward transform.

You can see the impulse as the small white square in the top-left corner of both images.

The wavenumber spectrum is flat

Because the wavenumber spectrum is flat, plots of the spectrum are completely featureless. Therefore, I did not include them in [Figure 5](#).

A very small error

The numeric output shows that the final output surface matches the input surface to within an error that is less than about one part in ten to the fourteenth power. The program produces the expected results for this test case.

If you were to go back to the equations in [Listing 2](#) and [Listing 3](#) and work this case out by hand, you would soon discover that the computational requirements are almost trivial. Most of the computation involves doing arithmetic using values of 1 and 0. Thus, there isn't a lot of opportunity for computational errors in this case.

Case 1

Now we are going to see a case that is more significant from a computational viewpoint. The input surface in this case will consist of a single impulse that is not located at the origin in the space domain. Rather, it is displaced from the origin.

The wavenumber amplitude spectrum of a single impulse in the space domain should be flat regardless of the location of the impulse in the space domain. However, the real and imaginary parts of the wavenumber spectrum are flat only when the impulse is located at the origin in space.

Regardless of the fact that the real and imaginary parts are not flat, the square root of the sum of the squares of the real and imaginary parts (*the amplitude*) should be the same for every point in wavenumber space for this case. Thus, the real and imaginary parts are related in a very special way.

The code for Case 1

The code that is executed when `switchCase` equals 1 is shown in [Listing 16](#).

Listing 16. Code for Case 1.

```
case 1:  
    spatialData[2][2] = 1;  
break;
```

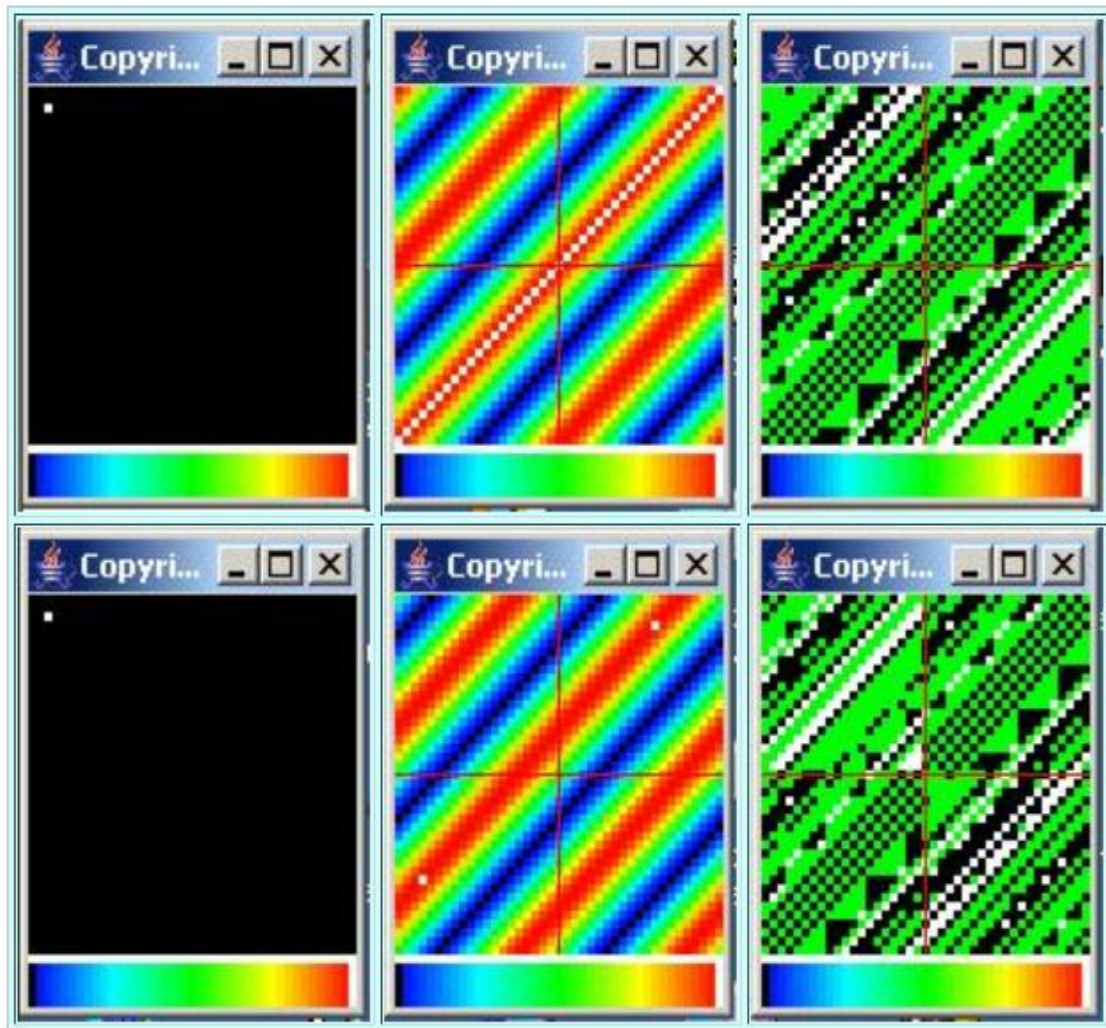
This case places a single impulse close to but not at the origin in space. This produces a flat amplitude spectrum in wavenumber space just like in Case 0. However, the real and imaginary parts of the spectrum are different from Case 0. They are not flat. The computations are probably more subject to errors in this case than for Case 0.

The visual output

[Figure 6](#) shows the six output images produced by the program for a **switchCase** value of 1.

Figure 6. A displaced impulse in space.

Figure 6. A displaced impulse in space.



The input and output surfaces match visually

The input and output surfaces showing the single impulse are in the two leftmost images in [Figure 6](#). From a visual viewpoint, the output at the bottom appears to be an exact match for the input at the top.

The real and imaginary parts

The real and imaginary parts of the wavenumber spectrum are shown in the two center images. The real part is at the top and the imaginary part is at the bottom.

For a single impulse in the space domain, we would expect each of these surfaces to be a 3D sinusoidal wave (*similar to a piece of corrugated sheet metal*) . That appears to be what we are seeing, with almost two full cycles of the sinusoidal wave between the origin and the bottom right corner of the image.

(The distance between the peaks in the sinusoidal wave in wavenumber space is inversely proportional to the distance of the impulse from the origin in space. Hence, as the impulse approaches the origin in space, the peaks in wavenumber space become further and further apart. When the impulse is located at the origin in space, the distance between the peaks in wavenumber space becomes infinite, leading to flat real and imaginary parts.)

Symmetry

We know that the real part of a wavenumber spectrum resulting from the Fourier transform of a real space function is symmetric about the origin. We also know that the imaginary part is asymmetric about the origin.

The symmetry/asymmetry requirements appear to be satisfied by this case. The color bands in the real part at the top are symmetric on either side of the origin.

The imaginary part is asymmetric about the origin (*the centers of the red/white and the blue/black bands appear to be equidistant from and on opposite sides of the origin*) .

The amplitude spectrum is ugly

The amplitude spectrum is shown in the two rightmost images in [Figure 6](#). The unshifted amplitude spectrum is shown at the top. The amplitude spectrum with the origin shifted to the center is shown at the bottom.

The ugliness of these two plots is an **artifact** of the 3D plotting scheme implemented by the class named **ImgMod29**. In order to maximize the use of the available dynamic range in the plot, each surface that is plotted is normalized such that:

- The highest elevation is colored white
- The lowest elevation is colored black
- Elevations between the highest and lowest values are colored according to the calibration scale below the image

This normalization is applied even when the distance between the highest and lowest elevation is very small. As a result of computational errors, the amplitude spectrum is not perfectly flat. Rather there are very small variations from one point to the next. As a result, the colors used to plot the surface switch among the full range of available colors even for tiny deviations from perfect flatness.

A very small error

The total error for this case is very small. The numeric output shows that the final output surface matches the input surface to within an error that is less than about one part in ten to the thirteenth power. The program produces the expected results for this test case.

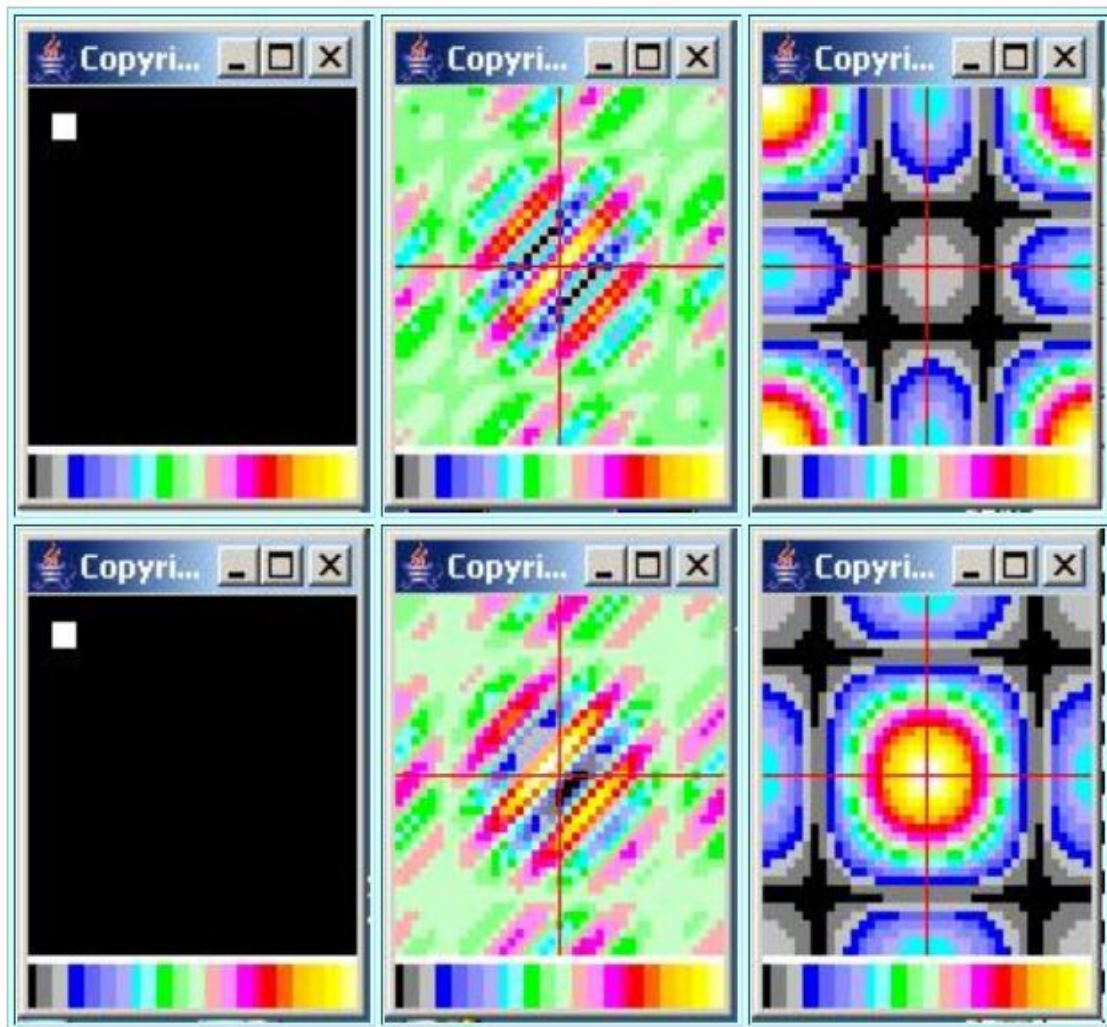
Case 2

Now we are going to take a look at another case for which we know in advance generally what the outcome should be. This will allow us to compare the outcome with our expectations to confirm proper operation of the program.

A box on the diagonal in space

This case places a box that is one unit tall on the diagonal near the origin in the space domain as shown in the top-left image in [Figure 7](#).

Figure 7. A box on the diagonal in space.



What do we know?

On the basis of prior experience, we know that the amplitude spectrum of this surface along the horizontal and vertical axes of the wavenumber spectrum should have a rectified $\sin(x)/x$ shape (*all negative values are converted to positive values*). We know that the peak in this amplitude spectrum should appear at the origin in wavenumber space, and that the width of the peak should be inversely proportional to the size of the box.

The code for Case 2

The code that constructs the space domain surface for this case is shown in [Listing 17](#).

Listing 17. Code for Case 2.

```
case 2:
    spatialData[3][3] = 1;
    spatialData[3][4] = 1;
    spatialData[3][5] = 1;
    spatialData[4][3] = 1;
    spatialData[4][4] = 1;
    spatialData[4][5] = 1;
    spatialData[5][3] = 1;
    spatialData[5][4] = 1;
    spatialData[5][5] = 1;
break;
```

This code is completely straightforward. It sets the value of each of nine adjacent points on the surface to a value of 1, while the values of all other

points on the surface remain at zero. The arrangement of those nine points forms a square whose sides are parallel to the horizontal and vertical axes.

The real and imaginary parts of the spectrum

There isn't a lot that I can tell you about what to expect regarding the real and imaginary parts of this spectrum, other than that they should exhibit the same [symmetry](#) and asymmetry conditions that I described earlier for the real and imaginary parts in general. These requirements appear to be satisfied by the real part at the top center of [Figure 7](#) and the imaginary part at the bottom center of [Figure 7](#).

Otherwise, the shape of the real and imaginary wavenumber spectra will depend on the location of the box in space and the size and orientation of the box.

A different plotting color scheme

Note that the plotting color scheme that I used for [Figure 7](#) is different from any of the plots previously shown in this module. This color scheme is what I refer to as the Color Contour scheme in the module titled [Plotting 3D Surfaces using Java](#).

This scheme quantizes the range from the lowest to the highest elevation into 23 levels, coloring the lowest elevation black, the highest elevation white, and assigning very specific colors to the 21 levels in between. The colors and the levels that they represent are shown in the calibration scales under the plots in [Figure 7](#). The lowest elevation is on the left end of the calibration scale. The highest elevation is on the right end of the calibration scale.

The amplitude spectrum

As before, the wavenumber amplitude spectrum with the origin in the top-left corner is shown in the top-right image in [Figure 7](#). The amplitude spectrum

with the origin shifted to the center is shown in the lower right image in [Figure 7](#).

If you were to use the calibration scale to convert the colors along the horizontal and vertical axes in the lower right image into numeric values, you would find that they approximate a rectified $\sin(x)/x$ shape as expected.

The output surface

The output surface produced by performing an inverse Fourier transform on the complex wavenumber spectrum is shown in the lower-left image in [Figure 7](#). This surface appears to match the input surface shown in the top-left image in [Figure 7](#).

The overall results

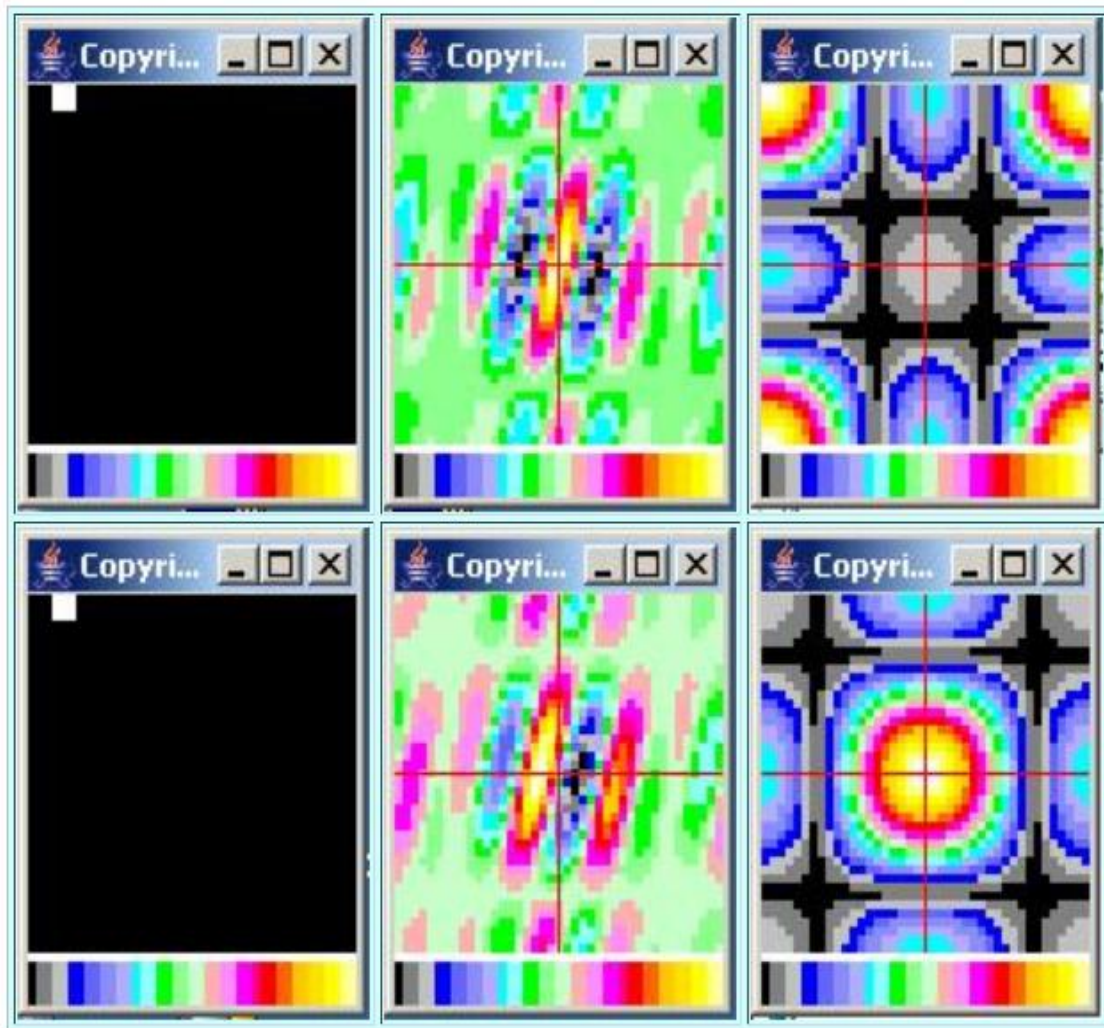
The numeric output for this case isn't very useful because none of the samples for which numeric data is provided fall within the square. However, because the real and imaginary parts exhibit the correct [symmetry](#), the shape of the amplitude spectrum is generally what we expect, and the output from the inverse Fourier transform appears to match the original input causes us to conclude that the program is working properly in this case.

Case 3

This case places a raised box at the top near the origin in the space domain, but the box is not on the diagonal as it was in Case 2.

(See the top-left image in [Figure 8](#) for the new location of the box.)

Figure 8. Graphic output for Case 3.



Amplitude spectrum should not change

As long as the size and the orientation of the box doesn't change, the wavenumber amplitude spectrum should be the same as Case 2 regardless of the location of the box in space. Since the size and orientation of this box is the same as in Case 2, the amplitude spectrum for this case should be the same as for Case 2.

The real and imaginary parts of the spectrum may change

However, the real and imaginary parts (*or the phase*) change as the location of the box changes relative to the origin in space.

A hypothetical example

The purpose of this case is to illustrate a hypothetical example. If two different photographic images contain a picture of the same object in the same size and the same orientation in space, that object will contribute the same values to the amplitude spectrum of both images regardless of where the object is located in the different images.

For example, assume that a photographic image includes a picture of a vase. Assume that the original image is cropped twice along two different borders producing two new images. Assume that both of the new images contain the picture of the vase, but in different locations. That vase will contribute the same values to the amplitude spectra of the two images regardless of the location of the vase in each of the images. This knowledge will be useful to us in future modules when we begin using 2D Fourier transforms to process photographic images.

Amplitude spectrum is the same

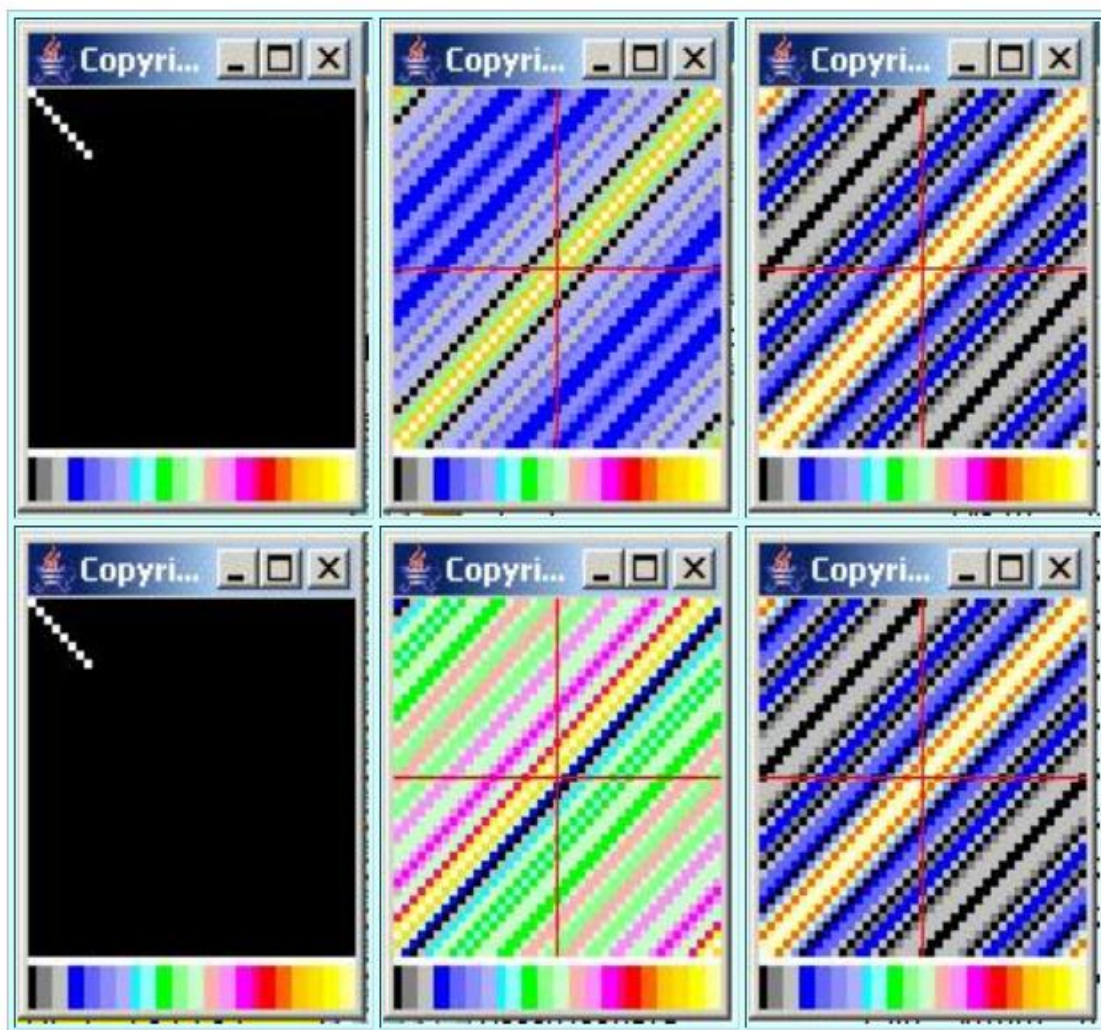
If you compare [Figure 8](#) with [Figure 7](#), you will see that the amplitude spectrum is the same for both surfaces despite the fact that the box is in a different location in each of the two surfaces. However, the real and imaginary parts of the spectrum in [Figure 8](#) are considerably different from the real and imaginary parts of the spectrum in [Figure 7](#).

The code that was used to create the surface for this case is straightforward. You can view that code in [Listing 22](#) near the end of the module.

Case 4

This case draws a short line containing eight points along the diagonal from top-left to lower right in the space domain. You can view this surface in the top-left image in [Figure 9](#). You can view the code that generated this surface in [Listing 22](#) near the end of the module.

Figure 9. Graphic output for Case 4.



Another example of $\sin(x)/x$

On the basis of prior experience, we would expect the wavenumber amplitude spectrum, (*when viewed along any line in wavenumber space parallel to the line in space*) , to have a rectified $\sin(x)/x$ shape. We would expect the peak of that shape to be centered on the origin in wavenumber space. We would expect the width of the peak in wavenumber space to be inversely proportional to the length of the line in space.

We would expect the amplitude spectrum when viewed along any line in wavenumber space perpendicular to the line in space to have a constant value.

Our expectations are borne out

The shape of the amplitude spectrum shown in the lower right image in [Figure 9](#) agrees with our expectations. Although not shown here, if we were to make the line of points longer, the width of the peak in the rectified $\sin(x)/x$ would become narrower. If we were to make the line of points shorter, the peak would become wider, as we will demonstrate in Case 5.

Our expectations regarding [symmetry](#) and asymmetry for the real and imaginary parts shown in the center images of [Figure 9](#) are borne out. The real part is at the top center and the imaginary part is at the bottom center.

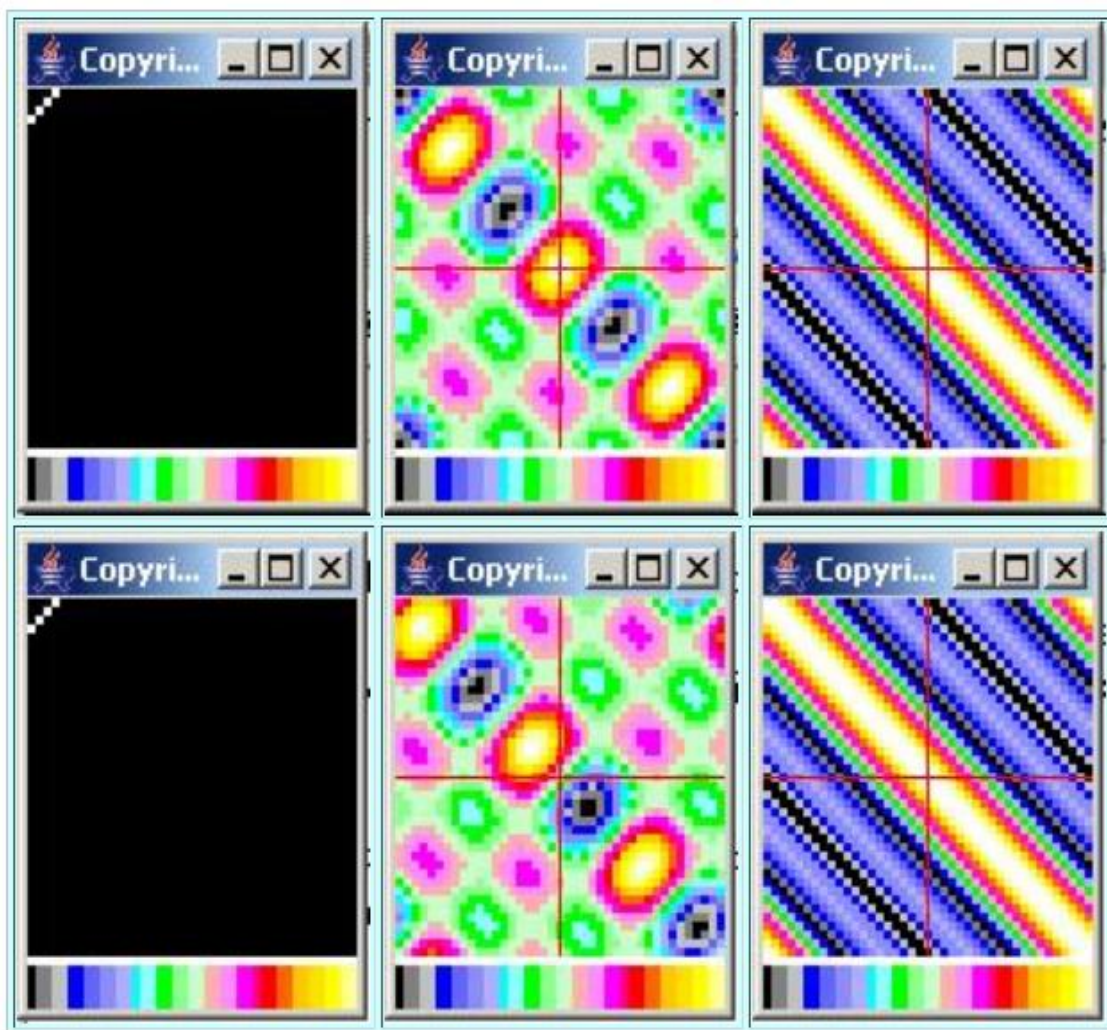
The output from the inverse Fourier transform shown in the bottom left of [Figure 9](#) matches the original space domain surface in the top left of [Figure 9](#).

Case 5

This case draws a short line consisting of only four points perpendicular to the diagonal from top-left to lower right. This line of points is perpendicular to the direction of the line of points in Case 4.

You can view the surface for this case in the top-left image of [Figure 10](#). You can view the code that generated this surface in [Listing 22](#) near the end of the module.

Figure 10. Graphic output for Case 5.



Rotated by ninety degrees

If you compare [Figure 10](#) with [Figure 9](#), you will see that the spectral result is rotated ninety degrees relative to that shown for Case 4 where the line was along the diagonal. In other words, rotating the line of points by ninety degrees also rotated the structure in the wavenumber spectrum by ninety degrees.

A wider peak

In addition, the line of points for Case 5 is shorter than the line of points for Case 4 resulting in a wider peak in the rectified $\sin(x)/x$ shape for Case 5.

The real and imaginary parts

While the real and imaginary parts of the spectrum shown in the center of [Figure 10](#) are considerably different from anything that we have seen prior to this, they still satisfy the [symmetry](#) and asymmetry conditions that we expect for the real and imaginary parts.

The final output matches the input

The output from the inverse Fourier transform in the bottom left image in [Figure 10](#) matches the input surface in the top left image in [Figure 10](#).

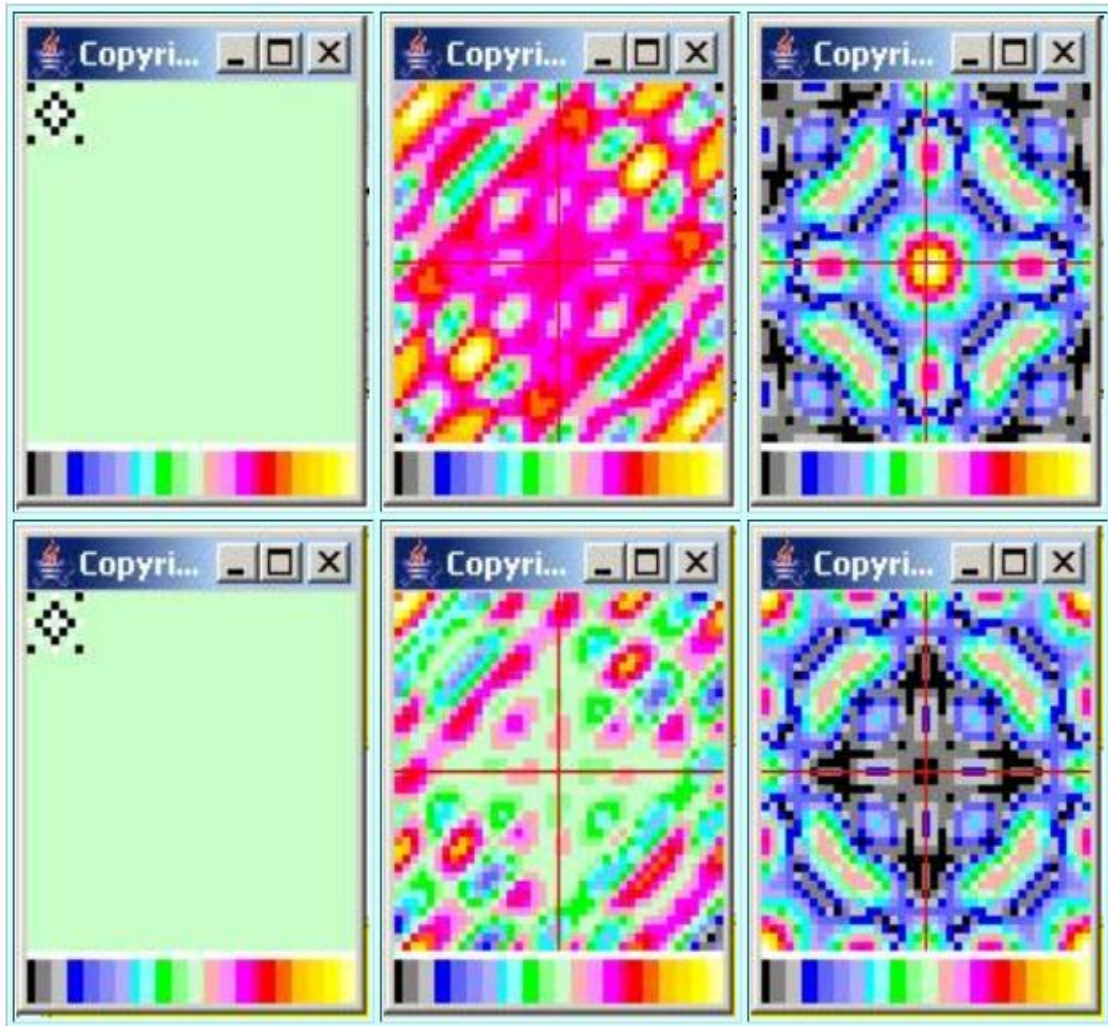
All of this matches our expectations for this case.

Case 6

This case is considerably more complicated than the previous cases. You can view the surface for this case in the top-left image in [Figure 11](#). You can view the code that generated this surface in [Listing 22](#) near the end of the module.

Figure 11. Graphic output for Case 6.

Figure 11. Graphic output for Case 6.



Many weighted lines of points

This case draws horizontal lines, vertical lines, and lines on both diagonals. Each individual point on each line is given a value of either +1 or -1. The weights of the individual points are adjusted so that the sum of all the weights is 0. The weight at the point where the lines intersect is also 0.

Black is -1, white is +1

The small black squares in the top-left image in [Figure 11](#) represent points with a weight of -1. The small white squares represent points with a weight of +1. The green background color represents a value of 0.

Symmetries on four different axes

The wavenumber amplitude spectrum is shown in the bottom right image in [Figure 11](#). As you can see from that image, performing a 2D Fourier transform on this surface produces a wavenumber amplitude spectrum that is symmetrical along lines drawn at 0, 45, 90, and 135 degrees to the horizontal. There is a line of symmetry in the amplitude spectrum for every line of points on the space domain surface.

Must be zero at the wavenumber origin

Because the sum of all the points is 0, the value of the wavenumber spectrum at the origin must also be zero. This is indicated by the black square at the origin in the lower right image.

Peaks at the folding wave numbers

This amplitude spectrum has major peaks at the folding wave number on each of the 45-degree axes. In addition, there are minor peaks at various other points in the spectrum.

The real and imaginary parts

As expected, the real and imaginary parts of the spectrum, shown in the center of [Figure 11](#) exhibit the required [symmetry](#) and asymmetry that I discussed earlier.

The final output

The output produced by performing an inverse Fourier transform on the complex wavenumber spectrum is shown in the lower-left image in [Figure 11](#). This image matches the input surface shown in the top left image in [Figure 11](#).

Case 7

Now we are going to make a major change in direction. All of the surfaces from cases 0 through 6 consisted of a few individual points located in specific geometries in the space domain. All of the remaining points on the surface had a value of zero. This resulted in continuous (*but sampled*) surfaces in the wavenumber domain.

Now we are going to generate continuous (*but sampled*) surfaces in the space domain. We will generate these surfaces as sinusoidal surfaces (*similar to a sheet of corrugated sheet metal*) or the sums of sinusoidal surfaces.

Performing Fourier transforms on these surfaces will produce amplitude spectra consisting of a few non-zero points in wavenumber space with the remaining points in the spectrum having values near zero.

Need to change the surface plotting scale

In order to make these amplitude spectra easier to view, I have modified the program to cause the square representing each point in the amplitude spectrum to be five pixels on each side instead of three pixels on each side. To keep the overall size of the images under control, I reduced the width and the height of the surfaces from 41 points to 23 points.

Display fewer results

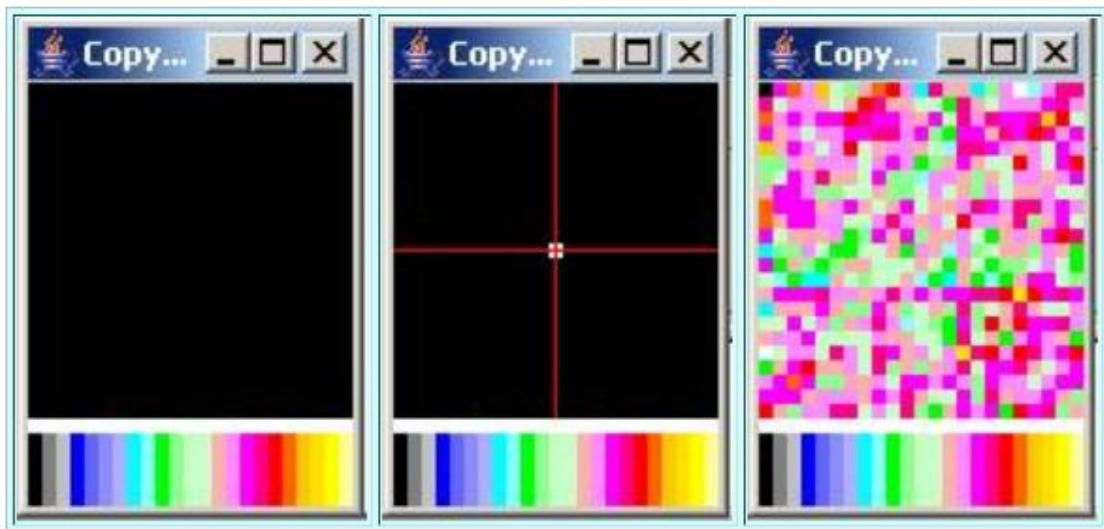
I suspect that you have seen all the real parts, imaginary parts, and unshifted amplitude spectra that you want to see. Therefore, at this point, I will begin displaying only the input surface, the amplitude spectrum, and the output

surface that results from performing an inverse Fourier transform on the complex spectrum.

A zero frequency sine wave

The first example in this category is shown in [Figure 12](#). The input surface for this example is a sinusoidal wave with a frequency of zero. This results in a perfectly flat surface in the space domain as shown in the leftmost image in [Figure 12](#). This surface is perfectly flat and featureless.

Figure 12. Graphic output for Case 7.



The code for this case

The code that was used to generate this surface is shown in [Listing 18](#). For the case of a sinusoidal wave with zero frequency, every point on the surface has a value of 1.0.

Listing 18. Code for Case 7.

```
case 7:
    for(int row = 0; row < rows; row++){
        for(int col = 0; col < cols; col++){
            spatialData[row][col] = 1.0;
        } //end inner loop
    } //end outer loop
    break;
```

A single point at the origin

As shown by the center image in [Figure 12](#), the Fourier transform of this surface produces a single point at the origin in wavenumber space. This is exactly what we would expect.

The inverse transform output is ugly

The result of performing an inverse Fourier transform on the complex spectrum is shown in the rightmost image in [Figure 12](#). As was the case earlier in [Figure 6](#), the ugliness of this plot is an artifact of the 3D plotting scheme implemented by the class named **ImgMod29**. The [explanation](#) that I gave there applies here also.

A very small error

Once again, the total error is very small. The numeric output shows that the final output surface matches the input surface to within an error that is less than about one part in ten to the thirteenth power. Thus, the program produces the expected results for this test case.

Case 8

This case draws a sinusoidal surface along the horizontal axis with one sample per cycle.

(This surface is under sampled by a factor of two under the commonly held belief that there should be at least two samples per cycle of the highest frequency component in the surface.)

Thus, it is impossible to distinguish this surface from a surface consisting of a sinusoid with a frequency of zero.

The code that was used to produce this surface is shown in [Listing 19](#). This code is typical of the code that I will be using to produce the remaining surfaces in this module. This code is straightforward and shouldn't require further explanation.

Listing 19. Code for Case 8.

Listing 19. Code for Case 8.

```
case 8:
    for(int row = 0; row < rows; row++){
        for(int col = 0; col < cols; col++){
            spatialData[row][col] =
cos(2*PI*col/1);
        }//end inner loop
    }//end outer loop
break;
```

The graphic output for Case 8

The Fourier transform of this surface produces a single peak at the origin in the wavenumber spectrum just like in [Figure 12](#). I didn't provide a display of the graphic output for this case because it looks just like the graphic output shown for the zero frequency sinusoid in [Figure 12](#).

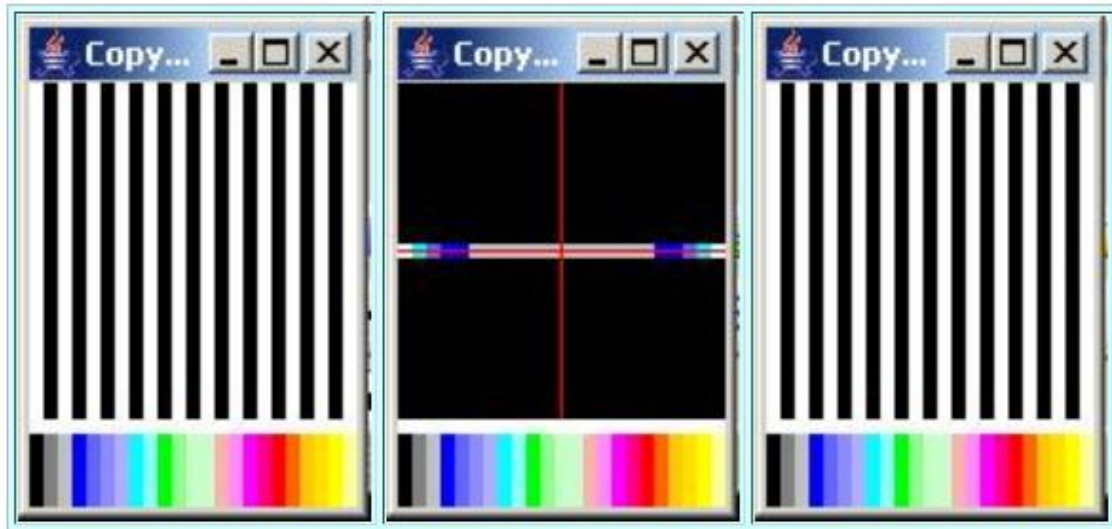
Case 9

This case draws a sinusoidal surface along the horizontal axis with two samples per cycle as shown in the leftmost image in [Figure 13](#). This corresponds to the Nyquist folding wavenumber.

The wavenumber spectrum

The center image in [Figure 13](#) shows the wavenumber amplitude spectrum for this surface. The wavenumber spectrum has white peak values at the positive and negative folding wave numbers on the right and left edges of the imaged. The colors in between these two peaks are green, blue, and gray indicating very low values.

Figure 13. Graphic output for Case 9.



The inverse Fourier transform output

The output from the inverse Fourier transform performed on the complex wavenumber spectrum for this case is shown in the rightmost image in [Figure 13](#). The output is a good match for the input shown on the left.

You can view the code that was used to create this surface in [Listing 22](#) near the end of the module.

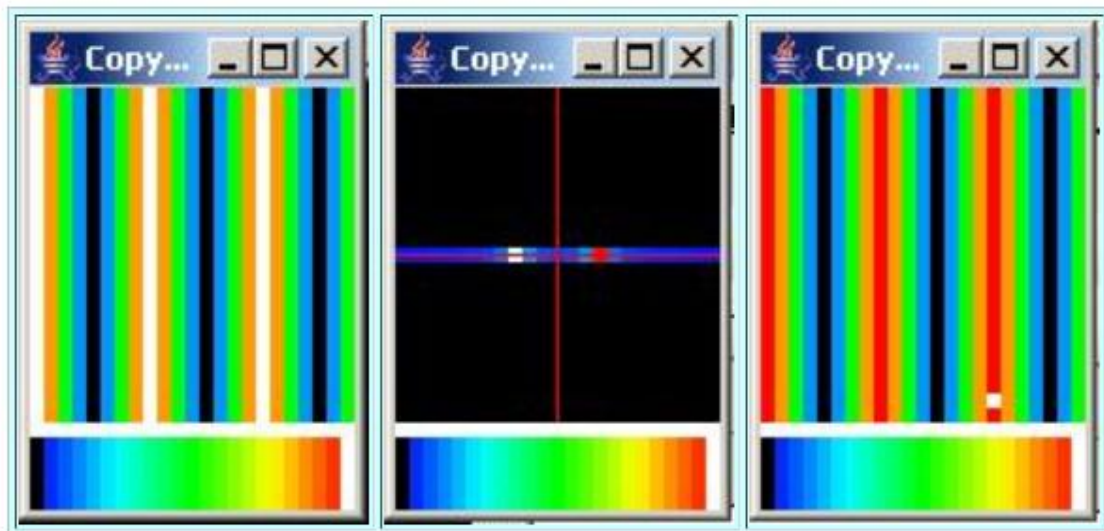
Case 10

This case draws a sinusoidal surface along the vertical axis with two samples per cycle. Again, this is the Nyquist folding wave number but the sinusoid appears along the vertical axis instead of appearing along the horizontal axis. If you run this case and view the results, you will see that it replicates the results from Case 9 except that everything is rotated by ninety degrees in both the space domain and the wavenumber domain.

Case 11

This case draws a sinusoidal surface along the horizontal axis with eight samples per cycle as shown in the leftmost image of [Figure 14](#).

Figure 14. Graphic output for Case 11.



The wavenumber spectrum

Performing a forward Fourier transform on this surface produces symmetrical peaks on the horizontal axis on either side of the wavenumber origin. The two peaks are indicated by the small white and red squares on the horizontal axis in the center image in [Figure 14](#).

*(Recall that for the plotting format used in [Figure 14](#), the color white is reserved for the single point with the highest elevation. The difference in an elevation colored white and an elevation colored red for this plotting format might be as small as one part in ten to the fourteenth or fifteenth power. As a practical matter, **red and white** indicate the same elevation for this plotting format.)*

For a sinusoidal surface with eight samples per cycle, we would expect the peaks to occur in the wavenumber spectrum about one-fourth of the distance from the origin to the folding wavenumber. [Figure 14](#) meets that expectation.

The peaks are surrounded on both sides by blue and cyan colors, indicating very low values.

The inverse Fourier transform output

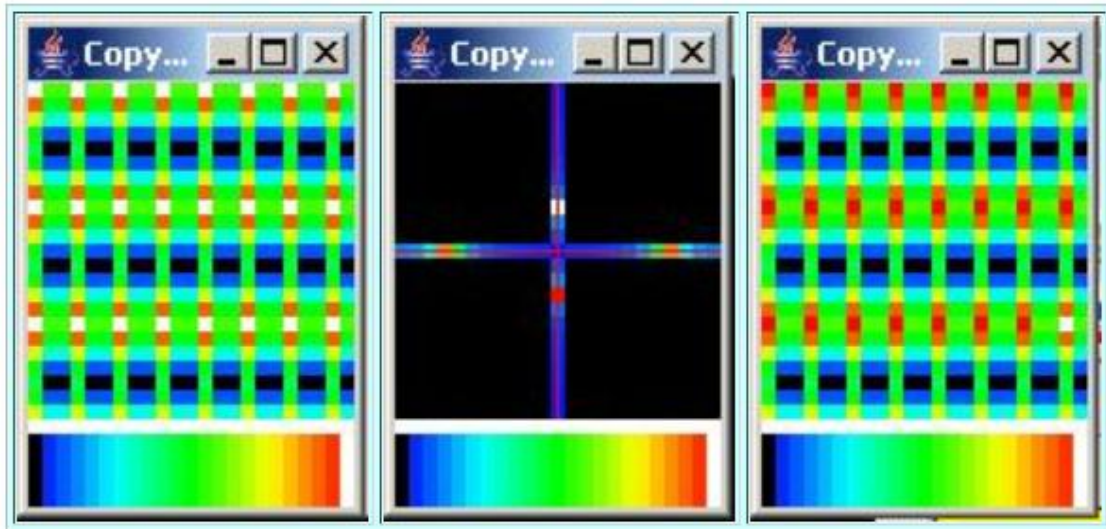
The output from the inverse Fourier transformed performed on the complex spectrum is shown in the rightmost image in [Figure 14](#). This output compares very favorably with the input surface shown in the leftmost image. The difference between the two is that the input has white vertical bands whereas the output has red vertical bands (*with a single white spot*). The above [explanation](#) of white versus red applies here also.

You can view the code that created this surface in [Listing 22](#) near the end of the module.

Case 12

This case draws a sinusoidal surface on the horizontal axis with three samples per cycle plus a sinusoidal surface on the vertical axis with eight samples per cycle as shown by the leftmost image in [Figure 15](#).

Figure 15. Graphic output for Case 12.



The wavenumber spectrum

Performing a forward Fourier transform produces symmetrical peaks on the horizontal and vertical axes on all four sides of the wave number origin. These peaks are indicated by the red and white squares in the center image in [Figure 15](#).

(See the earlier discussion regarding the difference in elevation indicated by [red and white](#) for this plotting format.)

The peaks on the vertical axis should be about one-fourth of the way between the origin and the folding wavenumber. This appears to be the case. The peaks on the horizontal axis should be about two-thirds of the way between the origin and the folding wavenumber, which they also appear to be.

Inverse Fourier transform output

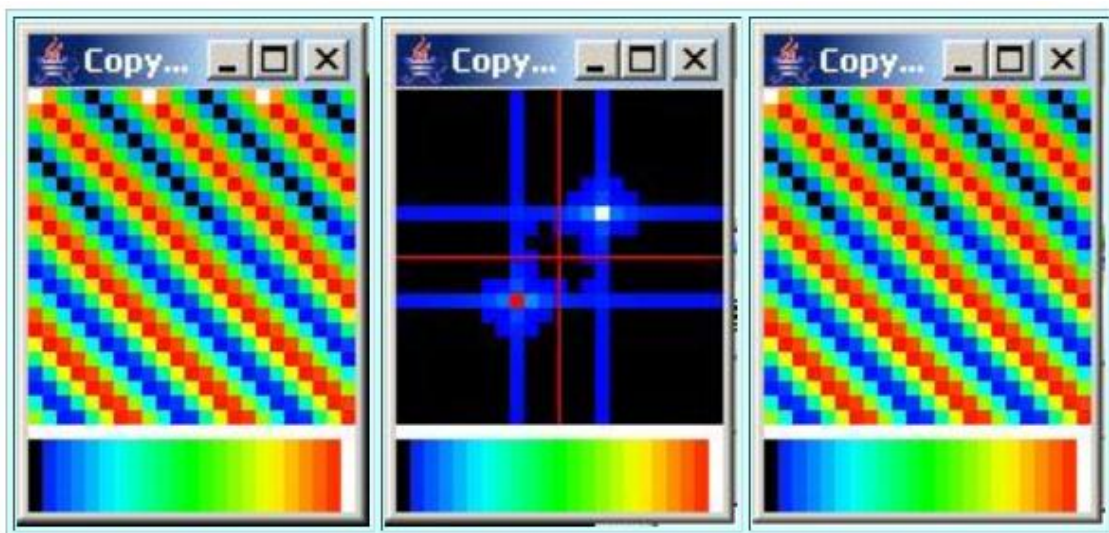
The output produced by performing an inverse Fourier transform on the complex spectrum is shown in the rightmost image in [Figure 15](#). Taking the [red versus white](#) issue into account, this output compares favorably with the input surface shown in the leftmost image in [Figure 15](#).

You can view the code that created this surface in [Listing 22](#) near the end of the module.

Case 13

This case draws a sinusoidal surface at an angle of approximately 45 degrees relative to the horizontal as shown in the leftmost image in [Figure 16](#). This sinusoid has approximately eight samples per cycle.

Figure 16. Graphic output for Case 13.



The wavenumber spectrum

Performing a forward Fourier transform on this surface produces a pair of peaks in the wavenumber spectrum that are symmetrical about the origin at approximately 45 degrees relative to the horizontal axis. These peaks are indicated by the red and white squares in the center image in [Figure 16](#).

The inverse Fourier transform output

The output produced by performing an inverse Fourier transform on the complex wavenumber spectrum is shown in the rightmost image in [Figure 16](#). This output compares favorably with the input surface shown in the leftmost image in [Figure 16](#).

You can view the code that created this surface in [Listing 22](#) near the end of the module.

The end of the `getSpatialData` method

[Listing 20](#) shows the end of the method named `getSpatialData` and the end of the class named `ImgMod31`.

Listing 20. The end of the `getSpatialData` method.

Listing 20. The end of the getSpatialData method.

```
        default:
            System.out.println("Case must be " +
                               "between 0 and 13
inclusive.");
            System.out.println(
                               "Terminating
program.");
            System.exit(0);
        }//end switch statement

        return spatialData;
    }//end getSpatialData
}//end class ImgMod31
```

A default case is provided in the switch statement to deal with the possibility that the user may specify a case that is not included in the allowable limits of 0 through 13 inclusive.

The method ends by returning a reference to the array object containing the 3D surface that was created by the selected case in the switch statement.

Run the program

I encourage you to copy, compile, and run the programs that you will find in [Listing 21](#) and [Listing 22](#) near the end of the module.

*(You will also need to go to the module titled [Plotting 3D Surfaces using Java](#) and get a copy of the source code for the program named **ImgMod29** .)*

Modify the programs and experiment with them in order to learn as much as you can about 2D Fourier transforms.

Create some different test cases and work with them until you understand why they produce the results that they do.

Summary

I began [Part 1](#) of this two-part series by explaining how the space domain and the wavenumber domain in two-dimensional analysis are analogous to the time domain and the frequency domain in one-dimensional analysis.

Then I introduced you to some practical examples showing how 2D Fourier transforms and wavenumber spectra can be useful in solving engineering problems involving antenna arrays.

In this module, I provided and explained a class that can be used to perform forward and inverse 2D Fourier transforms, and can also be used to shift the wavenumber origin from the top-left to the center for a more pleasing plot of the wavenumber spectral data.

Finally, I provided and explained a program that is used to:

- Test the forward and inverse 2D Fourier transforms to confirm that the code is correct and that the transforms behave as they should
- Produce wavenumber spectra for simple surfaces to help the student gain a feel for the relationships that exist between the space domain and the wavenumber domain

Complete program listings

Complete listings of the classes presented in this module are provided in [Listing 21](#) and [Listing 22](#) below.

Listings for other programs mentioned in the module, such as **Dsp029**, are provided in other modules. Those modules are identified in the text of this module.

Listing 21. ImgMod30.java.

```
/*File ImgMod30.java  
Copyright 2005, R.G.Baldwin
```

The purpose of this program is to provide 2D Fourier Transform capability to be used for image processing and other purposes. The class provides three static methods:

xform2D: Performs a forward 2D Fourier transform on a surface described by a 2D array of double values in the space domain to produce a spectrum in the wavenumber domain. The method returns the real part, the imaginary part, and the amplitude spectrum, each in its own 2D array of double values.

inverseXform2D: Performs an inverse 2D Fourier transform from the wavenumber domain into the space domain using the real and imaginary parts of the wavenumber spectrum as input. Returns the surface in the space domain in a 2D array of double values.

shiftOrigin: The wavenumber spectrum produced by xform2D has its origin in the top-left corner with the Nyquist folding wave numbers near the center. This is not a very suitable format for visual analysis. This method rearranges the data to place the origin at the center with the Nyquist folding wave numbers along the edges.

Tested using J2SE 5.0 and WinXP

```
*****/  
import static java.lang.Math.*;
```

Listing 21. ImgMod30.java.

```
class ImgMod30{

    //This method computes a forward 2D Fourier
    // transform from the space domain into the
    // wavenumber domain. The number of points
    // produced for the wavenumber domain matches
    // the number of points received for the space
    // domain in both dimensions. Note that the
    // input data must be purely real. In other
    // words, the program assumes that there are
    // no imaginary values in the space domain.
    // Therefore, it is not a general purpose 2D
    // complex-to-complex transform.
    static void xform2D(double[][] inputData,
                        double[][] realOut,
                        double[][] imagOut,
                        double[][] amplitudeOut){

        int height = inputData.length;
        int width = inputData[0].length;

        System.out.println("height = " + height);
        System.out.println("width = " + width);

        //Two outer loops iterate on output data.
        for(int yWave = 0;yWave < height;yWave++){
            for(int xWave = 0;xWave < width;xWave++){
                //Two inner loops iterate on input data.
                for(int ySpace = 0;ySpace < height;
                    ySpace++){
                    for(int xSpace = 0;xSpace < width;
                        xSpace++){
                        //Compute real, imag, and ampltude. Note that it
                        // was necessary to sacrifice indentation to
                        // force these very long equations to be
```

Listing 21. ImgMod30.java.

```
// compatible with this narrow publication format
// and still be somewhat readable.
realOut[yWave][xWave] +=
(inputData[ySpace][xSpace]*cos(2*PI*((1.0*
xWave*xSpace/width)+(1.0*yWave*ySpace/height))))
/sqrt(width*height);

imagOut[yWave][xWave] -=
(inputData[ySpace][xSpace]*sin(2*PI*((1.0*xWave*
xSpace/width) + (1.0*yWave*ySpace/height))))
/sqrt(width*height);

amplitudeOut[yWave][xWave] =
sqrt(
realOut[yWave][xWave] * realOut[yWave][xWave] +
imagOut[yWave][xWave] * imagOut[yWave][xWave]);
    }//end xSpace loop
  }//end ySpace loop
} //end xWave loop
} //end yWave loop
} //end xform2D method
//-----//

//This method computes an inverse 2D Fourier
// transform from the wavenumber domain into
// the space domain. The number of points
// produced for the space domain matches
// the number of points received for the wave-
// number domain in both dimensions. Note that
// this method assumes that the inverse
// transform will produce purely real values in
// the space domain. Therefore, in the
// interest of computational efficiency, it
// does not compute the imaginary output
// values. Therefore, it is not a general
// purpose 2D complex-to-complex transform. For
```

Listing 21. ImgMod30.java.

```
// correct results, the input complex data must
// match that obtained by performing a forward
// transform on purely real data in the space
// domain.

static void inverseXform2D(double[][] real,
                           double[][] imag,
                           double[][] dataOut){

    int height = real.length;
    int width = real[0].length;

    System.out.println("height = " + height);
    System.out.println("width = " + width);

    //Two outer loops iterate on output data.
    for(int ySpace = 0;ySpace < height;ySpace++){
        for(int xSpace = 0;xSpace < width;
              xSpace++){
            //Two inner loops iterate on input data.
            for(int yWave = 0;yWave < height;
                  yWave++){
                for(int xWave = 0;xWave < width;
                      xWave++){
                    //Compute real output data. Note that it was
                    // necessary to sacrifice indentation to force
                    // this very long equation to be compatible with
                    // this narrow publication format and still be
                    // somewhat readable.
                    dataOut[ySpace][xSpace] +=
                    (real[yWave][xWave]*cos(2*PI*((1.0 * xSpace*
                    xWave/width) + (1.0*ySpace*yWave/height))) -
                    imag[yWave][xWave]*sin(2*PI*((1.0 * xSpace*
                    xWave/width) + (1.0*ySpace*yWave/height))))
                    /sqrt(width*height);
```

Listing 21. ImgMod30.java.

```
        }//end xWave loop
    }//end yWave loop
} //end xSpace loop
} //end ySpace loop
} //end inverseXform2D method
//-----//

//Method to shift the wavenumber origin and
// place it at the center for a more visually
// pleasing display. Must be applied
// separately to the real part, the imaginary
// part, and the amplitude spectrum for a wave-
// number spectrum.
static double[][] shiftOrigin(double[][] data){
    int numberOfRows = data.length;
    int numberOfCols = data[0].length;
    int newRows;
    int newCols;

    double[][] output =
        new double[numberOfRows][numberOfCols];

    //Must treat the data differently when the
    // dimension is odd than when it is even.

    if(numberOfRows%2 != 0){//odd
        newRows = numberOfRows +
            (numberOfRows + 1)/2;
    }else{//even
        newRows = numberOfRows + numberOfRows/2;
    }//end else

    if(numberOfCols%2 != 0){//odd
        newCols = numberOfCols +
            (numberOfCols + 1)/2;
    }else{//even
```

Listing 21. ImgMod30.java.

```
        newCols = numberOfCols + numberOfCols/2;
    }//end else

    //Create a temporary working array.
    double[][] temp =
        new double[newRows][newCols];

    //Copy input data into the working array.
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;col < numberOfCols;col++){
            temp[row][col] = data[row][col];
        }//col loop
    }//row loop

    //Do the horizontal shift first
    if(numberOfCols%2 != 0){//shift for odd

        //Slide leftmost (numberOfCols+1)/2 columns
        // to the right by numberOfCols columns
        for(int row = 0;row < numberOfRows;row++){
            for(int col = 0;
                col < (numberOfCols+1)/2;col++){
                temp[row][col + numberOfCols] =
                    temp[row][col];
            }//col loop
        }//row loop

        //Now slide everything back to the left by
        // (numberOfCols+1)/2 columns
        for(int row = 0;row < numberOfRows;row++){
            for(int col = 0;
                col < numberOfCols;col++){
                temp[row][col] =
                    temp[row][col+(numberOfCols + 1)/2];
            }//col loop
        }//row loop
    }
```

Listing 21. ImgMod30.java.

```
}else{//shift for even
    //Slide leftmost (numberOfCols/2) columns
    // to the right by numberOfCols columns.
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
            col < numberOfCols/2;col++){
            temp[row][col + numberOfCols] =
                temp[row][col];
        }//col loop
    }//row loop

    //Now slide everything back to the left by
    // numberOfCols/2 columns
    for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
            col < numberOfCols;col++){
            temp[row][col] =
                temp[row][col + numberOfCols/2];
        }//col loop
    }//row loop
}//end else

//Now do the vertical shift
if(numberOfRows%2 != 0){//shift for odd
    //Slide topmost (numberOfRows+1)/2 rows
    // down by numberOfRows rows.
    for(int col = 0;col < numberOfCols;col++){
        for(int row = 0;
            row < (numberOfRows+1)/2;row++){
            temp[row + numberOfRows][col] =
                temp[row][col];
        }//row loop
    }//col loop

    //Now slide everything back up by
```

Listing 21. ImgMod30.java.

```
// (numberOfRows+1)/2 rows.
for(int col = 0; col < numberOfCols; col++){
    for(int row = 0;
        row < numberOfRows; row++){
        temp[row][col] =
            temp[row+(numberOfRows + 1)/2][col];
    }//row loop
}//col loop

}else{//shift for even
    //Slide topmost (numberOfRows/2) rows down
    // by numberOfRows rows
    for(int col = 0; col < numberOfCols; col++){
        for(int row = 0;
            row < numberOfRows/2; row++){
            temp[row + numberOfRows][col] =
                temp[row][col];
        }//row loop
    }//col loop

    //Now slide everything back up by
    // numberOfRows/2 rows.
    for(int col = 0; col < numberOfCols; col++){
        for(int row = 0;
            row < numberOfRows; row++){
            temp[row][col] =
                temp[row + numberOfRows/2][col];
        }//row loop
    }//col loop
}//end else

//Shifting of the origin is complete. Copy
// the rearranged data from temp to output
// array.
for(int row = 0; row < numberOfRows; row++){
    for(int col = 0; col < numberOfCols; col++){
```


Listing 21. ImgMod30.java.

```
        output[row][col] = temp[row][col];
    } // col loop
} // row loop

    return output;
} // end shiftOrigin method

} // end class ImgMod30
```

Listing 22. ImgMod31.java.

```
/*File ImgMod31.java
Copyright 2005, R.G.Baldwin
```

The purpose of this program is to exercise and test the 2D Fourier Transform methods and the axis shifting method provided by the class named ImgMod30.

The main method in this class reads a command-line parameter and uses it to select a specific case involving a particular kind of input data in the space domain. The program then performs a 2D Fourier transform on that data followed by an inverse 2D Fourier transform.

There are 14 cases built into the program with case numbers ranging from 0 to 13 inclusive. Each of the cases is designed such that the

Listing 22. ImgMod31.java.

results should be known in advance by a person familiar with 2D Fourier analysis and the wavenumber domain. The cases are also designed to illustrate the impact of various space-domain characteristics on the wavenumber spectrum. This information will be useful later when analyzing the results of performing 2D transforms on photographic images and other images as well.

Each time the program is run, it produces a stack of six output images in the top-left corner of the screen. The type of each image is listed below. This list is in top-to-bottom order. To view the images further down in the stack, you must physically move those on top to get them out of the way.

The top-to-bottom order of the output images is as follows:

1. Space-domain output of inverse Fourier transform. Compare with original input in 6 below.
2. Amplitude spectrum in wavenumber domain with shifted origin. Compare with 5 below.
3. Imaginary wavenumber spectrum with shifted origin.
4. Real wavenumber spectrum with shifted origin.
5. Amplitude spectrum in wavenumber domain without shifted origin. Compare with 2 above.
6. Space-domain input data. Compare with 1 above.

In addition, the program produces some numeric

Listing 22. ImgMod31.java.

output on the command-line screen that may be useful in confirming the validity of the inverse transform. The following is an example:

```
height = 41
width = 41
height = 41
width = 41
2.0 1.99999999999999916
0.50000000000000002 0.499999999999999845
0.49999999999999956 0.49999999999999923
1.7071067811865475 1.7071067811865526
0.2071067811865478 0.20710678118654233
0.20710678118654713 0.20710678118655435
1.0 1.00000000000000064
-0.49999999999999997 -0.499999999999999484
-0.50000000000000003 -0.49999999999999965
```

The first two lines above indicate the size of the spatial surface for the forward transform. The second two lines indicate the size of the wavenumber surface for the inverse transform.

The remaining nine lines indicate something about the quality of the inverse transform in terms of its ability to replicate the original spatial surface. These lines also indicate something about the correctness or lack thereof of the overall scaling from original input to final output. Each line contains a pair of values. The first value is from the original spatial surface. The second value is from the spatial surface produced by performing an inverse transform on the wavenumber spectrum. The two values in each pair of values should match. If they match, this indicates the probability of a

Listing 22. ImgMod31.java.

valid result. Note however that this is a very small sampling of the values that make up the original and replicated spatial data and problems could arise in areas that are not included in this small sample. The match is very good in the example shown above. This example is from Case #12.

Usage: java ImgMod31 CaseNumber DisplayType
CaseNumber from 0 to 13 inclusive.

If a case number is not provided, Case #2 will be run by default. If a display type is not provided, display type 1 will be used by default.

A description of each case is provided by the comments in this program.

See ImgMod29 for a definition of DisplayType, which can have a value of 0, 1, or 2.

You can terminate the program by clicking on the close button on any of the display frames produced by the program.

Tested using J2SE 5.0 and WinXP

```
*****/  
import static java.lang.Math.*;  
  
class ImgMod31{  
  
    public static void main(String[] args){  
        //Get input parameters to select the case to  
        // be run and the displayType. See ImgMod29  
        // for a description of displayType. Use  
        // default case and displayType if the user
```

Listing 22. ImgMod31.java.

```
// fails to provide that information.
// If the user provides a non-numeric input
// parameter, an exception will be thrown.
int switchCase = 2;//default
int displayType = 1;//default
if(args.length == 1){
    switchCase = Integer.parseInt(args[0]);
}else if(args.length == 2){
    switchCase = Integer.parseInt(args[0]);
    displayType = Integer.parseInt(args[1]);
}else{
    System.out.println("Usage: java ImgMod31 "
        + "CaseNumber DisplayType");
    System.out.println(
        "CaseNumber from 0 to 13 inclusive.");
    System.out.println(
        "DisplayType from 0 to 2 inclusive.");
    System.out.println("Running case "
        + switchCase + " by default.");
    System.out.println("Running DisplayType "
        + displayType + " by default.");
};//end else

//Create the array of test data.
int rows = 41;
int cols = 41;

//Get a test surface in the space domain.
double[][] spatialData =
    getSpatialData(switchCase,rows,cols);

//Display the spatial data.  Don't display
// the axes.
new ImgMod29(spatialData,3,false,
    displayType);
```

Listing 22. ImgMod31.java.

```
//Perform the forward transform from the
// space domain into the wavenumber domain.
// First prepare some array objects to
// store the results.
double[][] realSpect = //Real part
                        new double[rows][cols];
double[][] imagSpect = //Imaginary part
                        new double[rows][cols];
double[][] amplitudeSpect = //Amplitude
                             new double[rows][cols];
//Now perform the transform
ImgMod30.xform2D(spatialData,realSpect,
                  imagSpect,amplitudeSpect);

//Display the raw amplitude spectrum without
// shifting the origin first.  Display the
// axes.
new ImgMod29(amplitudeSpect,3,true,
              displayType);

//At this point, the wavenumber spectrum is
// not in a format that is good for viewing.
// In particular, the origin is at the top-
// left corner.  The horizontal Nyquist
// folding wavenumber is near the
// horizontal center of the plot.  The
// vertical Nyquist folding wave number is
// near the vertical center of the plot.  It
// is much easier for most people to
// understand what is going on when the
// wavenumber origin is shifted to the
// center of the plot with the Nyquist
// folding wave numbers at the edges of the
// plot.  The method named shiftOrigin can be
// used to rearrange the data and to shift
// the origin in that manner.
```

Listing 22. ImgMod31.java.

```
//Shift the origin and display the real part
// of the spectrum, the imaginary part of the
// spectrum, and the amplitude of the
// spectrum. Display the axes in all three
// cases.
double[][] shiftedRealSpect =
    ImgMod30.shiftOrigin(realSpect);
new ImgMod29(shiftedRealSpect,3,true,
    displayType);

double[][] shiftedImagSpect =
    ImgMod30.shiftOrigin(imagSpect);
new ImgMod29(shiftedImagSpect,3,true,
    displayType);

double[][] shiftedAmplitudeSpect =
    ImgMod30.shiftOrigin(amplitudeSpect);
new ImgMod29(shiftedAmplitudeSpect,3,true,
    displayType);

//Now test the inverse transform by
// performing an inverse transform on the
// real and imaginary parts produced earlier
// by the forward transform.
//Begin by preparing an array object to store
// the results.
double[][] recoveredSpatialData =
    new double[rows][cols];
//Now perform the inverse transform.
ImgMod30.inverseXform2D(realSpect,imagSpect,
    recoveredSpatialData);

//Display the output from the inverse
// transform. It should compare favorably
// with the original spatial surface.
```

Listing 22. ImgMod31.java.

```
new ImgMod29(recoveredSpatialData,3,false,
              displayType);

//Use the following code to confirm correct
// scaling. If the scaling is correct, the
// two values in each pair of values should
// match. Note that this is a very small
// subset of the total set of values that
// make up the original and recovered
// spatial data.
for(int row = 0;row < 3;row++){
    for(int col = 0;col < 3;col++){
        System.out.println(
            spatialData[row][col] + " " +
            recoveredSpatialData[row][col] + " ");
    }//col
};//row
};//end main
//=====//

//This method constructs and returns a 3D
// surface in a 2D array of type double
// according to the identification of a
// specific case received as an input
// parameter. There are 14 possible cases. A
// description of each case is provided in the
// comments. The other two input parameters
// specify the size of the surface in units of
// rows and columns.
private static double[][] getSpatialData(
    int switchCase,int rows,int cols){

    //Create an array to hold the data. All
    // elements are initialized to a value of
    // zero.
    double[][] spatialData =
```


Listing 22. ImgMod31.java.

```
new double[rows][cols];

//Use a switch statement to select and
// create a specified case.
switch(switchCase){
    case 0:
        //This case places a single non-zero
        // point at the origin in the space
        // domain. The origin is at the top-
        // left corner. In signal processing
        // terminology, this point can be viewed
        // as an impulse in space. This produces
        // a flat spectrum in wavenumber space.
        spatialData[0][0] = 1;
        break;

    case 1:
        //This case places a single non-zero
        // point near but not at the origin in
        // space. This produces a flat spectrum
        // in wavenumber space as in case 0.
        // However, the real and imaginary parts
        // of the transform are different from
        // case 0 and the result is subject to
        // arithmetic accuracy issues. The
        // plotted flat spectrum doesn't look
        // very good because the color switches
        // back and forth between three values
        // that are very close to together. This
        // is the result of the display program
        // normalizing the surface values based
        // on the maximum and minimum values,
        // which in this case are very close
        // together.
        spatialData[2][2] = 1;
        break;
```

Listing 22. ImgMod31.java.

```
case 2:
    //This case places a box on the diagonal
    // near the origin. This produces a
    //  $\sin(x)/x$  shape to the spectrum with
    // its peak at the origin in wavenumber
    // space.
    spatialData[3][3] = 1;
    spatialData[3][4] = 1;
    spatialData[3][5] = 1;
    spatialData[4][3] = 1;
    spatialData[4][4] = 1;
    spatialData[4][5] = 1;
    spatialData[5][3] = 1;
    spatialData[5][4] = 1;
    spatialData[5][5] = 1;
    break;

case 3:

    //This case places a box at the top near
    // the origin. This produces the same
    // amplitude spectrum as case 2. However,
    // the real and imaginary parts, (or the
    // phase) is different from case 2 due to
    // the difference in location of the box
    // relative to the origin in space.
    spatialData[0][3] = 1;
    spatialData[0][4] = 1;
    spatialData[0][5] = 1;
    spatialData[1][3] = 1;
    spatialData[1][4] = 1;
    spatialData[1][5] = 1;
    spatialData[2][3] = 1;
    spatialData[2][4] = 1;
    spatialData[2][5] = 1;
```

Listing 22. ImgMod31.java.

```
break;

case 4:
    //This case draws a short line along the
    // diagonal from top-left to lower
    // right. This results in a spectrum with
    // a sin(x)/x shape along that axis and a
    // constant along the axis that is
    // perpendicular to that axis
    spatialData[0][0] = 1;
    spatialData[1][1] = 1;
    spatialData[2][2] = 1;
    spatialData[3][3] = 1;
    spatialData[4][4] = 1;
    spatialData[5][5] = 1;
    spatialData[6][6] = 1;
    spatialData[7][7] = 1;
break;

case 5:
    //This case draws a short line
    // perpendicular to the diagonal from
    // top-left to lower right. The
    // spectral result is shifted 90 degrees
    // relative to that shown for case 4
    // where the line was along the diagonal.
    // In addition, the line is shorter
    // resulting in wider lobes in the
    // spectrum.
    spatialData[0][3] = 1;
    spatialData[1][2] = 1;
    spatialData[2][1] = 1;
    spatialData[3][0] = 1;
break;

case 6:
```

Listing 22. ImgMod31.java.

```
//This case draws horizontal lines,  
// vertical lines, and lines on both  
// diagonals. The weights of the  
// individual points is such that the  
// average of all the weights is 0.  
// The weight at the point where the  
// lines intersect is also 0. This  
// produces a spectrum that is  
// symmetrical across the axes at 0,  
// 45, and 90 degrees. The value of  
// the spectrum at the origin is zero  
// with major peaks at the folding  
// wavenumbers on the 45-degree axes.  
// In addition, there are minor peaks  
// at various other points as well.  
spatialData[0][0] = -1;  
spatialData[1][1] = 1;  
spatialData[2][2] = -1;  
spatialData[3][3] = 0;  
spatialData[4][4] = -1;  
spatialData[5][5] = 1;  
spatialData[6][6] = -1;  
  
spatialData[6][0] = -1;  
spatialData[5][1] = 1;  
spatialData[4][2] = -1;  
spatialData[3][3] = 0;  
spatialData[2][4] = -1;  
spatialData[1][5] = 1;  
spatialData[0][6] = -1;  
  
spatialData[3][0] = 1;  
spatialData[3][1] = -1;  
spatialData[3][2] = 1;  
spatialData[3][3] = 0;  
spatialData[3][4] = 1;
```

Listing 22. ImgMod31.java.

```
        spatialData[3][5] = -1;
        spatialData[3][6] = 1;

        spatialData[0][3] = 1;
        spatialData[1][3] = -1;
        spatialData[2][3] = 1;
        spatialData[3][3] = 0;
        spatialData[4][3] = 1;
        spatialData[5][3] = -1;
        spatialData[6][3] = 1;
    break;

    case 7:
        //This case draws a zero-frequency
        // sinusoid (DC) on the surface with an
        // infinite number of samples per cycle.
        // This causes a single peak to appear in
        // the spectrum at the wavenumber
        // origin. This origin is the top-left
        // corner for the raw spectrum, and is
        // at the center cross hairs after the
        // origin has been shifted to the
        // center for better viewing.
        for(int row = 0; row < rows; row++){
            for(int col = 0; col < cols; col++){
                spatialData[row][col] = 1.0;
            }//end inner loop
        }//end outer loop
    break;

    case 8:
        //This case draws a sinusoidal surface
        // along the horizontal axis with one
        // sample per cycle. This function is
        // under-sampled by a factor of 2.
        // This produces a single peak in the
```

Listing 22. ImgMod31.java.

```
// spectrum at the wave number origin.
// The result is the same as if the
// sinusoidal surface had zero frequency
// as in case 7..
for(int row = 0; row < rows; row++){
    for(int col = 0; col < cols; col++){
        spatialData[row][col] =
            cos(2*PI*col/1);
    }//end inner loop
}//end outer loop
break;

case 9:
    //This case draws a sinusoidal surface on
    // the horizontal axis with 2 samples per
    // cycle. This is the Nyquist folding
    // wave number. This causes a single
    // peak to appear in the spectrum at the
    // negative folding wave number on the
    // horizontal axis. A peak would also
    // appear at the positive folding wave
    // number if it were visible, but it is
    // one unit outside the boundary of the
    // plot.
    for(int row = 0; row < rows; row++){
        for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                cos(2*PI*col/2);
        }//end inner loop
    }//end outer loop
    break;

case 10:
    //This case draws a sinusoidal surface on
    // the vertical axis with 2 samples per
    // cycle. Again, this is the Nyquist
```

Listing 22. ImgMod31.java.

```
// folding wave number but the sinusoid
// appears along a different axis. This
// causes a single peak to appear in the
// spectrum at the negative folding wave
// number on the vertical axis. A peak
// would also appear at the positive
// folding wave number if it were
// visible, but it is one unit outside
// the boundary of the plot.
for(int row = 0; row < rows; row++){
    for(int col = 0; col < cols; col++){
        spatialData[row][col] =
                                cos(2*PI*row/2);
    }//end inner loop
}//end outer loop
break;

case 11:
    //This case draws a sinusoidal surface on
    // the horizontal axis with 8 samples per
    // cycle. You might think of this surface
    // as resembling a sheet of corrugated
    // roofing material. This produces
    // symmetrical peaks on the horizontal
    // axis on either side of the wave-
    // number origin.
    for(int row = 0; row < rows; row++){
        for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                                cos(2*PI*col/8);
        }//end inner loop
    }//end outer loop
    break;

case 12:
    //This case draws a sinusoidal surface on
```

Listing 22. ImgMod31.java.

```
// the horizontal axis with 3 samples per
// cycle plus a sinusoidal surface on the
// vertical axis with 8 samples per
// cycle. This produces symmetrical peaks
// on the horizontal and vertical axes on
// all four sides of the wave number
// origin.
for(int row = 0; row < rows; row++){
    for(int col = 0; col < cols; col++){
        spatialData[row][col] =
            cos(2*PI*row/8) + cos(2*PI*col/3);
    }//end inner loop
}//end outer loop
break;
```

case 13:

```
//This case draws a sinusoidal surface at
// an angle of approximately 45 degrees
// relative to the horizontal. This
// produces a pair of peaks in the
// wavenumber spectrum that are
// symmetrical about the origin at
// approximately 45 degrees relative to
// the horizontal axis.
double phase = 0;
for(int row = 0; row < rows; row++){
    for(int col = 0; col < cols; col++){
        spatialData[row][col] =
            cos(2.0*PI*col/8 - phase);
    }//end inner loop
    //Increase phase for next row
    phase += .8;
}//end outer loop
break;
```

default:

Listing 22. ImgMod31.java.

```
        System.out.println("Case must be " +  
                           "between 0 and 13 inclusive.");  
        System.out.println(  
                           "Terminating program.");  
        System.exit(0);  
    }//end switch statement  
  
    return spatialData;  
}//end getSpatialData  
}//end class ImgMod31
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1491-2D Fourier Transforms using Java, Part 2
- File: Java1491.htm
- Published: 08/09/05

Examine the code for a Java class that can be used to perform forward and inverse 2D Fourier transforms on 3D surfaces in the space domain. Learn how the 2D Fourier transform behaves for a variety of different sample surfaces in the space domain.

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be

aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1487-Convolution and Frequency Filtering in Java

Learn how to take advantage of time-domain convolution for frequency filtering using Java.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *Convolution and Frequency Filtering in Java* that was published by Prof. Baldwin around 2005. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if the tutorial refers to another tutorial or to a program that is not included, try

searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1487-Convolution and Frequency Filtering in Java
- File: Java1487.htm
- Published: 04/19/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1488-Convolution and Matched Filtering in Java

Learn how to perform matched filtering in Java. Learn how matched filtering often makes it possible to detect properly designed signals in noisy environments where simple frequency filtering alone cannot do the job.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named ***Convolution and Matched Filtering in Java*** that was published by Prof. Baldwin around 2005. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if

the tutorial refers to another tutorial or to a program that is not included, try searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1488-Convolution and Matched Filtering in Java
- File: Java1488.htm
- Published: 04/19/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2350-Adaptive Filtering in Java, Getting Started

Learn how to write a Java program to adaptively design a time-delay convolution filter with a flat amplitude response and a linear phase response using an LMS adaptive algorithm.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *Adaptive Filtering in Java, Getting Started* that was published by Prof. Baldwin around 2005. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if

the tutorial refers to another tutorial or to a program that is not included, try searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2350-Adaptive Filtering in Java, Getting Started
- File: Java2350.htm
- Published: 04/21/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2352-An Adaptive Whitening Filter in Java

Learn how to write an adaptive whitening filter program in Java. Also learn how to use the whitening filter to extract wide-band signal that is corrupted by one or more components of narrow-band noise.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *An Adaptive Whitening Filter in Java* that was published by Prof. Baldwin around 2005. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if

the tutorial refers to another tutorial or to a program that is not included, try searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2352-An Adaptive Whitening Filter in Java
- File: Java2352.htm
- PPublished: 04/24/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2354-A General-Purpose LMS Adaptive Engine in Java
Learn how to write a general-purpose LMS adaptive engine in Java, and how to demonstrate the use of the engine for three different adaptive programs of increasing complexity.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *A General-Purpose LMS Adaptive Engine in Java* that was published by Prof. Baldwin around 2005. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if

the tutorial refers to another tutorial or to a program that is not included, try searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2354-A General-Purpose LMS Adaptive Engine in Java
- File: Java2354.htm
- Published: 04/24/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2356-An Adaptive Line Tracker in Java

Learn how to use a general-purpose LMS adaptive engine to write an adaptive spectral line tracker in Java.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *An Adaptive Line Tracker in Java* that was published by Prof. Baldwin around 2005. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if the tutorial refers to another tutorial or to a program that is not included, try

searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2356-An Adaptive Line Tracker in Java
- File: Java2356.htm
- Published: 04/24/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2358-Adaptive Identification and Inverse Filtering using Java
Learn how to write a Java program that illustrates adaptive identification filtering and adaptive inverse filtering. Exercise the program for different scenarios.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *Adaptive Identification and Inverse Filtering using Java* that was published by Prof. Baldwin around 2006. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if

the tutorial refers to another tutorial or to a program that is not included, try searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2358-Adaptive Identification and Inverse Filtering using Java
- File: Java2358.htm
- Published: 04/24/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2360-Adaptive Noise Cancellation using Java

Learn how to use a general-purpose LMS adaptive engine to write a Java program that illustrates the use of adaptive filtering for noise cancellation.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named ***Adaptive Noise Cancellation using Java*** that was published by Prof. Baldwin around 2006. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if the tutorial refers to another tutorial or to a program that is not included, try

searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2360-Adaptive Noise Cancellation using Java
- File: Java2360.htm
- Published: 04/25/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2362-Adaptive Prediction using Java

Learn how to use a Java adaptive filter to predict future values in a time series. Discover the relationship between the properties of the time series and the quality of the prediction.

Table of contents

- [Preface](#)
- [Tutorial links](#)
- [Legacy content and references](#)
- [Legacy versus openstax presentation format](#)
- [Miscellaneous](#)

Preface

This is a cover page for a tutorial named *Adaptive Prediction using Java* that was published by Prof. Baldwin around 2006. Some things don't change with time and the contents of the tutorial are as relevant today as when the tutorial was originally published.

Tutorial Links

You can access the tutorial in either HTML or PDF format by clicking on one of the links below:

- [HTML](#)
- [PDF](#)

*(See the caution below regarding HTML and the **openstax** presentation format.)*

Legacy content and references

The tutorial may contain references to source code that is defined in other tutorials. Prof. Baldwin has attempted to make certain that all of the source code required by each tutorial is contained within that tutorial. However, if

the tutorial refers to another tutorial or to a program that is not included, try searching for it by title on cnx.org or on the web. It is probably available somewhere.

The tutorial may contain internal links to other tutorials that Prof. Baldwin has written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the referenced tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial on cnx.org.

By publishing the tutorial on the **openstax CNX** site, the tutorial is being licensed under a Creative Commons Attribution 4.0 License even though the copyright notice on the original tutorial document may be more restrictive.

Legacy versus openstax presentation format

Early in 2014, cnx.org began a transition from a [legacy presentation format](#) to a new [openstax](#) presentation format. As of April 7, 2014, some of the functionality of the legacy presentation format that is required by this module had not yet been ported to the **openstax** presentation format. *(In particular, image files referenced by hyperlinks in the HTML version of the tutorial may not display properly in the **openstax** presentation format.)*

This issue should be resolved at some point in the future. In the meantime, one of your options is to select and view the PDF version of the tutorial using the PDF link provided above.

A second option is to click the **Legacy Site** link at the top of this page *(assuming that you are not already on the Legacy Site)* and view the tutorial in its original HTML format. *(The HTML format is more reliable than the PDF format, particularly with regard to source code listings.)*

Later, when the issue mentioned above is resolved, you can select either the PDF version or the HTML version directly from the **openstax** presentation page, whichever you prefer.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2362-Adaptive Prediction using Java
- File: Java2362.htm
- Published: 04/25/14

Note: Disclaimers:

Financial : Although the **openstax CNX** site makes it possible for you to download a PDF file for the collection that contains this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

You also need to know that Prof. Baldwin receives no financial compensation from **openstax CNX** even if you purchase the PDF version of the collection.

In the past, unknown individuals have copied Prof. Baldwin's modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing Prof. Baldwin as the author. Prof. Baldwin neither receive compensation for those sales nor does he know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a collection that is freely available on **openstax CNX** and that it was made and published without the prior knowledge of Prof. Baldwin.

Affiliation : Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2440 Understanding the Lempel-Ziv Data Compression Algorithm in Java

Learn how to write a Java program that illustrates lossless data compression according to the Lempel-Ziv Compression Algorithm commonly known as LZ77. Also learn about the characteristics of the algorithm that result in data compression.

Table of contents

- [Preface](#)
- [Tutorial and code links](#)
- [Miscellaneous](#)

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org.

In the meantime, this is one of the pages in a book titled [Digital Signal Processing - DSP](#) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org.

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org. If not, you can probably use a [Google Advanced Search](#) to find a copy somewhere on the web.

Tutorial and code links

Click [here](#) to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click [here](#) to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2440 Understanding the Lempel-Ziv Data Compression Algorithm in Java
- File: Java2440.htm
- Published: 01/06/16

Note: Disclaimers:

Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced [above](#).

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.

In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on [cnx.org](#) and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2442 Understanding the Huffman Data Compression Algorithm in Java
Learn how to write a Java program that exposes the inner workings of the Huffman lossless data compression algorithm. Apply the algorithm to different test messages.

Table of contents

- [Preface](#)
- [Tutorial and code links](#)
- [Miscellaneous](#)

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org.

In the meantime, this is one of the pages in a book titled [Digital Signal Processing - DSP](#) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org.

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org. If not, you can probably use a [Google Advanced Search](#) to find a copy somewhere on the web.

Tutorial and code links

Click [here](#) to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click [here](#) to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2442 Understanding the Huffman Data Compression Algorithm in Java
- File: Java2442.htm
- Published: 01/06/16

Note: Disclaimers:

Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced [above](#).

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.

In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on [cnx.org](#) and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2444 Understanding the Discrete Cosine Transform in Java
Learn the basics of the Discrete Cosine Transform, which is used in many applications, including JPEG image compression.

Table of contents

- [Preface](#)
- [Tutorial and code links](#)
- [Miscellaneous](#)

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org.

In the meantime, this is one of the pages in a book titled [Digital Signal Processing - DSP](#) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org.

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org. If not, you can probably use a [Google Advanced Search](#) to find a copy somewhere on the web.

Tutorial and code links

Click [here](#) to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click [here](#) to download a zip file containing a clean copy of the

program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2444 Understanding the Discrete Cosine Transform in Java
- File: Java2444.htm
- Published: 01/06/16

Note: Disclaimers:

Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced [above](#).

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.

In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2446 Understanding the 2D Discrete Cosine Transform in Java
Learn how to use the forward two-dimensional Discrete Cosine Transform (2D-DCT) to compute and display the wave-number spectrum of an image. Also learn how to apply the inverse 2D-DCT to the spectral data to reconstruct and display a replica of the original image.

Table of contents

- [Preface](#)
- [Tutorial and code links](#)
- [Miscellaneous](#)

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org.

In the meantime, this is one of the pages in a book titled [Digital Signal Processing - DSP](#) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org.

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org. If not, you can probably use a [Google Advanced Search](#) to find a copy somewhere on the web.

Tutorial and code links

Click [here](#) to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click [here](#) to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2446 Understanding the 2D Discrete Cosine Transform in Java
- File: Java2446.htm
- Published: 01/06/16

Note: Disclaimers:

Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced [above](#).

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.

In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on [cnx.org](#) and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java2448 Understanding the 2D Discrete Cosine Transform in Java, Part 2
Learn how to sub-divide an image before applying a forward and inverse 2D-Discrete Cosine Transform similar to the way it is done in the JPEG image compression algorithm. Also learn some of the theory behind and some of the reasons for sub-dividing images, such as improved speed.

Table of contents

- [Preface](#)
- [Tutorial and code links](#)
- [Miscellaneous](#)

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org.

In the meantime, this is one of the pages in a book titled [Digital Signal Processing - DSP](#) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org.

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org. If not, you can probably use a [Google Advanced Search](#) to find a copy somewhere on the web.

Tutorial and code links

Click [here](#) to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click [here](#) to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java2448 Understanding the 2D Discrete Cosine Transform in Java, Part 2
- File: Java2448.htm
- Published: zz01/06/16

Note: Disclaimers:

Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced [above](#).

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.

In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on [cnx.org](#) and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-